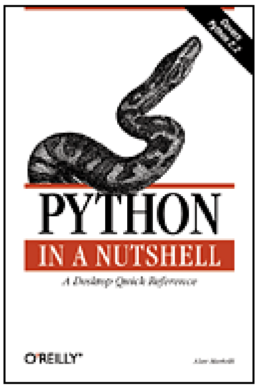
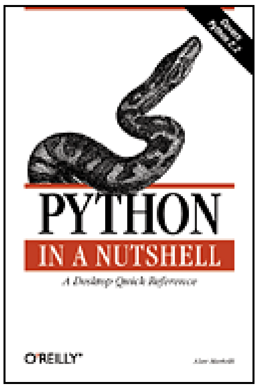


Python in a Nutshell

By Alex Martelli

Publisher : O'Reilly
Pub Date : March 2003
ISBN : 0-596-00188-6
Pages : 654

In the tradition of O'Reilly's "In a Nutshell" series, Python in a Nutshell offers Python programmers one place to look when they need help remembering or deciphering the syntax of this open source language and its many modules. This comprehensive reference guide makes it easy to look up all the most frequently needed information--not just about the Python language itself, but also the most frequently used parts of the standard library and the most important third-party extensions.



Python in a Nutshell

By Alex Martelli

Publisher	: O'Reilly
Pub Date	: March 2003
ISBN	: 0-596-00188-6
Pages	: 654

Copyright

Published by O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Preface

The Python programming language manages to reconcile many apparent contradictions: it's both elegant and pragmatic, simple and powerful, a high-level language that doesn't get in your way when you want to fiddle with bits and bytes, suitable for programming novices and great for experts too.

This book is aimed at programmers with some previous exposure to Python, as well as experienced programmers coming to Python for the first time from other programming languages. The book is a quick reference to Python itself, the most important parts of its vast standard library, and some of the most popular and useful third-party modules, covering a range of applications including web and network programming, GUIs, XML handling, database interactions, and high-speed numeric computing. It focuses on Python's cross-platform capabilities and covers the basics of extending Python and embedding it in other applications, using either C or Java.

How This Book Is Organized

This book has five parts, as follows:

Part I, Getting Started with Python

- Chapter 1 covers the general characteristics of the Python language and its implementations, and discusses where to get help and information.
- Chapter 2 explains how to obtain and install Python.
- Chapter 3 covers the Python interpreter program, its command-line options, and its use for running Python programs and in interactive sessions. The chapter also mentions text editors that are particularly suitable for editing Python programs, and examines some full-fledged integrated development environments, including IDLE, which comes free with standard Python.

Part II, Core Python Language and Built-ins

- Chapter 4 covers Python syntax, built-in data types, expressions, statements, and how to write and call functions.
- Chapter 5 explains object-oriented programming in Python.
- Chapter 6 covers how to deal with errors and abnormal conditions in Python programs.
- Chapter 7 covers the ways in which Python lets you group code into modules and packages, and how to define and import modules.
- Chapter 8 is a reference to built-in data types and functions, and some of the most fundamental modules in the standard Python library.
- Chapter 9 covers Python's powerful string-processing facilities, including regular expressions.

Part III, Python Library and Extension Modules

- Chapter 10 explains how to deal with files and text processing using built-in Python file objects, modules from Python's standard library, and platform-specific extensions for rich text I/O.

Conventions Used in This Book

The following conventions are used throughout this book.

Reference Conventions

In the function/method reference entries, when feasible, each optional parameter is shown with a default value using the Python syntax *name=value*. Built-in functions need not accept named parameters, so parameter names are not significant. Some optional parameters are best explained in terms of their presence or absence, rather than through default values. In such cases, a parameter is indicated as being optional by enclosing it in brackets ([ ]). When more than one argument is optional, the brackets are nested.

Typographic Conventions

Italic

Used for filenames, program names, URLs, and to introduce new terms.

`Constant Width`

Used for all code examples, as well as for commands and all items that appear in code, including keywords, methods, functions, classes, and modules.

`Constant Width Italic`

Used to show text that can be replaced with user-supplied values in code examples.

`Constant Width Bold`

Used for commands that must be typed on the command line, and occasionally for emphasis in code examples or to indicate code output.

How to Contact Us

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

O'Reilly & Associates 1005 Gravenstein Highway North Sebastopol, CA 95472 (800) 928-9938 (in the United States or Canada) (707) 829-0515 (international or local) (707) 829-0104 (fax)

There is a web page for this book, which lists errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/pythonian/

To ask technical questions or comment on the book, send email to:

bookquestions@oreilly.com

For more information about books, conferences, resource centers, and the O'Reilly Network, see the O'Reilly web site at:

http://www.oreilly.com

Acknowledgments

My heartfelt thanks to everybody who helped me out on this book. Many Python beginners, practitioners, and experts have read drafts of parts of the book and have given me feedback to help make it clearer and more precise, accurate, and readable. Out of those, for the quality and quantity of their feedback, I must single out for special thanks Andrea Babini, Andrei Raevsky, Anna Ravenscroft, and my fellow Python Business Forum board members Jacob Hallén and Laura Creighton.

Some Python experts gave me indispensable help in specific areas: Aahz on threading, Itamar Shtull-Trauring on Twisted, Mike Orr on Cheetah, Eric Jones and Paul Dubois on Numeric, and Tim Peters on threading, testing, performance issues, and optimization.

I was also blessed with a wonderful group of technical reviewers: Fred Drake of Python Labs, co-author of Python & XML (O'Reilly) and Grand Poobah of Python's excellent free documentation; Magnus Lie Hetland, author of Practical Python (Apress); Steve Holden, author of Python Web Programming (New Riders); and last but not least Sue Giller, whose observations as a sharp-eyed, experienced, non-Pythonista programmer were particularly useful in the pursuit of clarity and precision. The book's editor, Paula Ferguson, went above and beyond the call of duty in her work to make this book clearer and more readable.

My family and friends have been patient and supportive throughout the time it took me to write this book: particular thanks for that to my children Flavia and Lucio, my partner Marina, my sister Elisabetta, and my father Lanfranco.

Part I: Getting Started with Python

Chapter 1. Introduction to Python

Python is a general-purpose programming language. It has been around for quite a while: Guido van Rossum, Python's creator, started developing Python back in 1990. This stable and mature language is very high level, dynamic, object-oriented, and cross-platform—all characteristics that are very attractive to developers. Python runs on all major hardware platforms and operating systems, so it doesn't constrain your platform choices.

Python offers high productivity for all phases of the software life cycle: analysis, design, prototyping, coding, testing, debugging, tuning, documentation, deployment, and, of course, maintenance. Python's popularity has seen steady, unflagging growth over the years. Today, familiarity with Python is an advantage for every programmer, as Python is likely to have some useful role to play as a part of any software solution.

Python provides a unique mix of elegance, simplicity, and power. You'll quickly become productive with Python, thanks to its consistency and regularity, its rich standard library, and the many other modules that are readily available for it. Python is easy to learn, so it is quite suitable if you are new to programming, yet at the same time it is powerful enough for the most sophisticated expert.

1.1 The Python Language

The Python language, while not minimalist, is rather spare, for good pragmatic reasons. When a language offers one good way to express a design idea, supplying other ways has only modest benefits, while the cost in terms of language complexity grows with the number of features. A complicated language is harder to learn and to master (and to implement efficiently and without bugs) than a simpler one. Any complications and quirks in a language hamper productivity in software maintenance, particularly in large projects, where many developers cooperate and often maintain code originally written by others.

Python is simple, but not simplistic. It adheres to the idea that if a language behaves a certain way in some contexts, it should ideally work similarly in all contexts. Python also follows the principle that a language should not have convenient shortcuts, special cases, ad hoc exceptions, overly subtle distinctions, or mysterious and tricky under-the-covers optimizations. A good language, like any other designed artifact, must balance such general principles with taste, common sense, and a high degree of practicality.

Python is a general-purpose programming language, so Python's traits are useful in any area of software development. There is no area where Python cannot be part of an optimal solution. "Part" is an important word here—while many developers find that Python fills all of their needs, Python does not have to stand alone. Python programs can cooperate with a variety of other software components, making it an ideal language for gluing together components written in other languages.

Python is a very-high-level language. This means that Python uses a higher level of abstraction, conceptually farther from the underlying machine, than do classic compiled languages, such as C, C++, and Fortran, which are traditionally called high-level languages. Python is also simpler, faster to process, and more regular than classic high-level languages. This affords high programmer productivity and makes Python an attractive development tool. Good compilers for classic compiled languages can often generate binary machine code that runs much faster than Python code. However, in most cases, the performance of Python-coded applications proves sufficient. When it doesn't, you can apply the optimization techniques covered in Chapter 17 to enhance your program's performance while keeping the benefits of high programming productivity.

Python is an object-oriented programming language, but it lets you develop code using both object-oriented and traditional procedural styles, mixing and matching as your application requires. Python's object-oriented features are like those of C++, although they are much simpler to use.

1.2 The Python Standard Library and Extension Modules

There is more to Python programming than just the Python language: the standard Python library and other extension modules are almost as important for effective Python use as the language itself. The Python standard library supplies many well-designed, solid, 100% pure Python modules for convenient reuse. It includes modules for such tasks as data representation, string and text processing, interacting with the operating system and filesystem, and web programming. Because these modules are written in Python, they work on all platforms supported by Python.

Extension modules, from the standard library or from elsewhere, let Python applications access functionality supplied by the underlying operating system or other software components, such as graphical user interfaces (GUIs), databases, and networks. Extensions afford maximal speed in computationally intensive tasks, such as XML parsing and numeric array computations. Extension modules that are not coded in Python, however, do not necessarily enjoy the same cross-platform portability as pure Python code.

You can write special-purpose extension modules in lower-level languages to achieve maximum performance for small, computationally intensive parts that you originally prototyped in Python. You can also use tools such as SWIG to make existing C/C++ libraries into Python extension modules, as we'll see in Chapter 24. Finally, you can embed Python in applications coded in other languages, exposing existing application functionality to Python scripts via dedicated Python extension modules.

This book documents many modules, both from the standard library and from other sources, in areas such as client- and server-side network programming, GUIs, numerical array processing, databases, manipulation of text and binary files, and interaction with the operating system.

1.3 Python Implementations

Python currently has two production-quality implementations, CPython and Jython, and one experimental implementation, Python .NET. This book primarily addresses CPython, which I refer to as just Python for simplicity. However, the distinction between a language and its implementations is an important one.

1.3.1 CPython

Classic Python (a.k.a., CPython, often just called Python) is the fastest, most up-to-date, most solid and complete implementation of Python. CPython is a compiler, interpreter, and set of built-in and optional extension modules, coded in standard C. CPython can be used on any platform where the C compiler complies with the ISO/IEC 9899:1990 standard (i.e., all modern, popular platforms). In Chapter 2, I'll explain how to download and install CPython. All of this book, except Chapter 24 and a few sections explicitly marked otherwise, applies to CPython.

1.3.2 Jython

Jython is a Python implementation for any Java Virtual Machine (JVM) compliant with Java 1.2 or better. Such JVMs are available for all popular, modern platforms. To use Jython well, you need some familiarity with fundamental Java classes. You do not have to code in Java, but documentation and examples for existing Java classes are couched in Java terms, so you need a nodding acquaintance with Java to read and understand them. You also need to use Java supporting tools for tasks such as manipulating *.jar* files and signing applets. This book deals with Python, not with Java. For Jython usage, you should complement this book with Jython Essentials, by Noel Rappin and Samuele Pedroni (O'Reilly), possibly Java in a Nutshell, by David Flanagan (O'Reilly), and, if needed, some of the many other Java resources available.

1.3.3 Choosing Between CPython and Jython

If your platform is able to run both CPython and Jython, how do you choose between them? First of all, don't choose—download and install them both. They coexist without problems, and they're free. Having them both on your machine costs only some download time and a little extra disk space.

To experiment, learn, and try things out, you will most often use CPython, as it's faster. To develop and deploy, your best choice depends on what extension modules you want to use and how you want to distribute your programs. CPython applications are generally faster, particularly if they can make good use of suitable extension modules, such as Numeric (covered in Chapter 15). The development of CPython versions is faster than that of Jython versions: at the time of writing, for example, the next scheduled release is 2.2 for Jython, but 2.3 for CPython.

However, as you'll see in Chapter 25, Jython can use any Java class as an extension module, whether the class comes from a standard Java library, a third-party library, or a library you develop yourself. A Jython-coded application is a 100% pure Java application, with all of Java's deployment advantages and issues, and runs on any target machine having a suitable JVM. Packaging opportunities are also identical to Java's.

Jython and CPython are both good, faithful implementations of Python, reasonably close in terms of usability and performance. Given these pragmatic issues, either one may enjoy decisive practical advantages in a specific scenario. Thus, it is wise to become familiar with the strengths and weaknesses of each, to be able to choose optimally for each development task.

1.3.4 Python .NET

1.4 Python Development and Versions

Python is developed by the Python Labs of Zope Corporation, which consists of half a dozen core developers headed by Guido van Rossum, Python's inventor, architect, and Benevolent Dictator For Life (BDFL). This title means that Guido has the final say on what becomes part of the Python language and standard libraries.

Python intellectual property is vested in the Python Software Foundation (PSF), a non-profit corporation devoted to promoting Python, with dozens of individual members (nominated for their contributions to Python, and including all of the Python core team) and corporate sponsors. Most PSF members have commit privileges to Python's CVS tree on SourceForge (http://sf.net/cvs/?group_id=5470), and most Python CVS committers are members of the PSF.

Proposed changes to Python are detailed in public documents called Python Enhancement Proposals (PEPs), debated (and sometimes advisorily voted upon) by Python developers and the wider Python community, and finally approved or rejected by Guido, who takes debate and votes into account but is not bound by them. Hundreds of people contribute to Python development, through PEPs, discussion, bug reports, and proposed patches to Python sources, libraries, and documentation.

Python Labs releases minor versions of Python (2.x, for growing values of x) about once or twice a year. 2.0 was released in October 2000, 2.1 in April 2001, and 2.2 in December 2001. Python 2.3 is scheduled to be released in early 2003. Each minor release adds features that make Python more powerful and simpler to use, but also takes care to maintain backward compatibility. One day there will be a Python 3.0 release, which will be allowed to break backward compatibility to some extent. However, that release is still several years in the future, and no specific plans for it currently exist.

Each minor release 2.x starts with alpha releases, tagged as 2.xa0, 2.xa1, and so on. After the alphas comes at least one beta release, 2.xb1, and after the betas at least one release candidate, 2.xrc1. By the time the final release of 2.x comes out, it is always solid, reliable, and well tested on all major platforms. Any Python programmer can help ensure this by downloading alphas, betas, and release candidates, trying them out on existing Python programs, and filing bug reports for any problem that might emerge.

Once a minor release is out, most of the attention of the core team switches to the next minor release. However, a minor release normally gets successive point releases (i.e., 2.x.1, 2.x.2 and so on) that add no functionality but can fix errors, port Python to new platforms, enhance documentation, and add optimizations and tools.

The Python Business Forum (http://python-in-business.org) is an international society of companies that base their business on Python. The Forum, among other activities, tests and maintains special Python releases (known as "Python-in-a-tie") that Python Labs certifies for industrial-strength robustness.

This book focuses on Python 2.2 (and all its point releases), the most stable and widespread release at the time of this writing, and the basis of the current "Python-in-a-tie" efforts. It also mentions a few changes scheduled to appear in Python 2.3, and documents the parts of the language and libraries that are new in 2.2 and thus cannot be used with the previous 2.1 release. Python 2.1 is still important because it's used in widely deployed Zope 2.x releases (the current Zope releases, 3.x, rely on Python 2.2 and later). Also, at the time of this writing, the released version of Jython supports only Python 2.1, not yet Python 2.2.

Among older releases of Python, the only one with a large installed base is 1.5.2, which is part of most installations of Red Hat Linux Releases 6.x and 7.x. However, this book does not address Python 1.5.2, which is over three years

1.5 Python Resources

The richest of all Python resources is the Internet. The starting point is Python's site, http://www.python.org, which is full of interesting links that you will want to explore. And http://www.jython.org is a must if you have any interest in Jython.

1.5.1 Documentation

Python and Jython come with good documentation. The manuals are available in many formats, suitable for viewing, searching, and printing. You can browse the manuals on the Web at http://www.python.org/doc/current/. You can find links to the various formats you can download at http://www.python.org/doc/current/download.html, and http://www.python.org/doc/ has links to a large variety of documents. For Jython, http://www.jython.org/docs/ has links to Jython-specific documents as well as general Python ones. The Python FAQ (Frequently Asked Questions) is at http://www.python.org/doc/FAQ.html, and the Jython-specific FAQ is at http://www.jython.org/cgi-bin/faqw.py?req=index.

Most Python documentation (including this book) assumes some software development knowledge. However, Python is quite suitable for first-time programmers, so there are exceptions to this rule. A few good introductory online texts are:

- Josh Cogliati's "Non-Programmers Tutorial For Python," available at http://www.honors.montana.edu/~jjc/easytut/easytut/
- Alan Gauld's "Learning to Program," available at http://www.crosswinds.net/~agauld/
- Allen Downey and Jeffrey Elkner's "How to Think Like a Computer Scientist (Python Version)," available at http://www.ibiblio.org/obp/thinkCSpy/

1.5.2 Newsgroups and Mailing Lists

The URL http://www.python.org/psa/MailingLists.html has links to Python-related mailing lists and newsgroups. Always use plain-text format, not HTML, in all messages to mailing lists and newsgroups.

The Usenet newsgroup for Python discussions is comp.lang.python. The newsgroup is also available as a mailing list. To subscribe, send a message whose body is the word subscribe to python-list-request@python.org. Python-related announcements are posted to comp.lang.python.announce. To subscribe to its mailing-list equivalent, send a message whose body is the word subscribe to python-announce-list-request@python.org. To subscribe to Jython's mailing list, visit http://lists.sf.net/lists/listinfo/jython-users. To ask for individual help with Python, email your question to python-help@python.org. For questions and discussions about using Python to teach or learn programming, write to tutor@python.org.

1.5.3 Special Interest Groups

Chapter 2. Installation

You can install Python, in both classic (CPython) and JVM (Jython) versions, on most platforms. With a suitable development system (C for CPython, Java for Jython), you can install Python from its source code distribution. On popular platforms, you also have the alternative of installing from a prebuilt binary distribution.

Installing CPython from a binary distribution is faster, saves you substantial work on some platforms, and is the only possibility if you have no suitable C development system. Installing from a source code distribution gives you more control and flexibility, and is the only possibility if you can't find a suitable prebuilt binary distribution for your platform. Even if you install from binaries, I recommend you also download the source distribution, which includes examples and demos that may be missing from prebuilt binary packages.

2.1 Installing Python from Source Code

To install Python from source code, you need a platform with an ISO-compliant C compiler and ancillary tools such as *make*. On Windows, the normal way to build Python is with the Microsoft product Visual C++.

To download Python source code, visit http://www.python.org and follow the link labeled Download. The latest version at the time of this writing is:
http://www.python.org/ftp/python/2.2.2/Python-2.2.2.tgz

The *.tgz* file extension is equivalent to *.tar.gz* (i.e., a *tar* archive of files, compressed by the powerful and popular *gzip* compressor).

2.1.1 Windows

On Windows, installing Python from source code can be a chore unless you are already familiar with Microsoft Visual C++ and used to working at the Windows command line (i.e., in the text-oriented windows known as MS-DOS Prompt or Command Prompt, depending on your version of Windows).

If the following instructions give you trouble, I suggest you skip ahead to the material on installing Python from binaries later in this chapter. It may be a good idea, on Windows, to do an installation from binaries anyway, even if you also install from source code. This way, if you notice anything strange while using the version you installed from source code, you can double-check with the installation from binaries. If the strangeness goes away, it must have been due to some quirk in your installation from source code, and then you know you must double-check the latter.

In the following sections, for clarity, I assume you have made a new directory named *C:\Py* and downloaded *Python-2.2.2.tgz* there. Of course, you can choose to name and place the directory as it best suits you.

2.1.1.1 Uncompressing and unpacking the Python source code

You can uncompress and unpack a *.tgz* file with programs *tar* and *gunzip*. If you do not have *tar* and *gunzip*, you can download the collection of utilities ftp://ftp.objectcentral.com/winutils.zip into *C:\Py*. If you do not have other ways to unpack a ZIP file, download ftp://ftp.th-soft.com/UNZIP.EXE into *C:\Py*. Open an MS-DOS Prompt window and give the following commands:

```
 C:\> My Documents>cd \Py
C:\Py> unzip winutils
    [unzip lists the files it is unpacking - omitted here]
C:\Py> gunzip Python-2.2.2.tgz
C:\Py> tar xvf Python-2.2.2.tar
    [tar lists the files it is unpacking - omitted here]
C:\Py>
```

Commercial programs WinZip (http://www.winzip.com) and PowerArchiver (http://www.powerarchiver.com) can also uncompress and unpack *.tgz* archives. Whether via *gunzip* and *tar*, a commercial program, or some other program, you now have a directory *C:\Py\Python-2.2.2*, the root of a tree that contains the entire standard Python distribution in source form.

2.1.1.2 Building the Python source code with Microsoft Visual C++

Open the workspace file *C:\Py\Python-2.2.2\PCbuild\pcbuild.dsw* with Microsoft Visual C++, for example by starting Windows Explorer, going to directory *C:\Py\Python-2.2.2\PCbuild*, and double-clicking on file *pcbuild.dsw*

2.2 Installing Python from Binaries

If your platform is popular and current, you may find a prebuilt and packaged binary version of Python ready for installation. Binary packages are typically self-installing, either directly as executable programs, or via appropriate system tools, such as the RedHat Package Manager (RPM) on Linux and the Microsoft Installer (MSI) on Windows. Once you have downloaded a package, install it by running the program and interactively choosing installation parameters, such as the directory where Python is to be installed.

To download Python binaries, visit http://www.python.org and follow the link labeled Download. At the time of this writing, the only binary installer directly available from the main Python site is a Windows installer executable: http://www.python.org/ftp/python/2.2.2/Python-2.2.2.exe

Many third parties supply free binary Python installers for other platforms. For Linux distributions, see http://rpmfind.net if your distribution is RPM-based (RedHat, Mandrake, SUSE, and so on) or http://www.debian.org for Debian. The site http://www.python.org/download/ provides links to binary distributions for Macintosh, OS/2, Amiga, RISC OS, QNX, VxWorks, IBM AS/400, Sony PlayStation 2, and Sharp Zaurus. Older Python versions, mainly 1.5.2, are also usable and functional, though not as powerful and polished as the current Python 2.2.2. The download page provides links to 1.5.2 installers for older or less popular platforms (MS-DOS, Windows 3.1, Psion, BeOS, etc.).

ActivePython (http://www.activestate.com/Products/ActivePython) is a binary package of Python 2.2 for 32-bit versions of Windows and x86 Linux.

2.3 Installing Jython

To install Jython, you need a Java Virtual Machine (JVM) that complies with Java 1.1 or higher. See http://www.jython.org/platform.html for advice on JVMs for your platform.

To download Jython, visit http://www.jython.org and follow the link labeled Download. The latest version at the time of this writing is:
http://prdownloads.sf.net/jython/jython-21.class

In the following section, for clarity, I assume you have created a new directory named *C:\Jy* and downloaded *jython-21.class* there. Of course, you can choose to name and place the directory as it best suits you. On Unix-like platforms, in particular, the directory name will more likely be something like *~/Jy*.

The Jython installer *.class* file is a self-installing program. Open an MS-DOS Prompt window (or a shell prompt on a Unix-like platform), change directory to *C:\Jy*, and run your Java interpreter on the Jython installer. Make sure to include directory *C:\Jy* in the Java CLASSPATH. With most releases of Sun's Java Development Kit (JDK), for example, you can run:

```
C:\Jy> java -cp . jython-21
```

This runs a GUI installer that lets you choose destination directory and options. If you want to avoid the GUI, you can use the -o switch on the command line. The switch lets you specify the installation directory and options directly on the command line. For example:

```
C:\Jy> java -cp . jython-21 -o C:\Jython-2.1 demo lib source
```

installs Jython, with all optional components (demos, libraries, and source code), in directory *C:\Jython-2.1*. The Jython installation builds two small, useful command files. One, run as *jython* (named *jython.bat* on Windows), runs the interpreter. The other, run as *jythonc*, compiles Python source into JVM bytecode. You can add the Jython installation directory to your PATH, or copy these command files into any directory on your PATH.

You may want to use Jython with different JDKs on the same machine. For example, while JDK 1.4 is best for most development, you may also need to use JDK 1.1 occasionally in order to compile applets that can run on browsers that support only Java 1.1. In such cases, you could share a single Jython installation among multiple JVMs. However, to avoid confusion and accidents, I suggest you perform separate installations from the same Jython download on each JVM you want to support. Suppose, for example, that you have JDK 1.4 installed in *C:\Jdk14* and JDK 1.1 installed in *C:\Jdk11*. In this case, you could use the commands:

```
C:\Jy> \Jdk14\java -cp . jython-21 -o C:\Jy21-14 demo lib source
C:\Jy> \Jdk11\java -cp . jython-21 -o C:\Jy21-11 demo lib source
```

With these installations, you could then choose to work off *C:\Jy21-14* most of the time (e.g., by placing it in your PATH), and *cd* to *C:\Jy21-11* when you specifically need to compile applets with JDK 1.1.

Chapter 3. The Python Interpreter

To develop software systems in Python, you produce text files that contain Python source code and documentation. You can use any text editor, including those in Integrated Development Environments (IDEs). You then process the source files with the Python compiler and interpreter. You can do this directly, or implicitly inside an IDE, or via another program that embeds Python. The Python interpreter also lets you execute Python code interactively, as do IDEs.

3.1 The python Program

The Python interpreter program is run as *python* (it's named *python.exe* on Windows). *python* includes both the interpreter itself and the Python compiler, which is implicitly invoked, as needed, on imported modules. Depending on your system, the program may have to be in a directory listed in your PATH environment variable. Alternatively, as with any other program, you can give a complete pathname to it at the command (shell) prompt, or in the shell script (or *.BAT* file, shortcut target, etc.) that runs it.[1] On Windows, you can also use Start ⟶ Programs ⟶ Python 2.2 ⟶ Python (command line).

[1] This may involve using quotes, if the pathname contains spaces—again, this depends on your operating system.

3.1.1 Environment Variables

Besides PATH, other environment variables affect the *python* program. Some environment variables have the same effects as options passed to *python* on the command line; these are documented in the next section. A few provide settings not available via command-line options:

PYTHONHOME

The Python installation directory. A *lib* subdirectory, containing the standard Python library modules, should exist under this directory. On Unix-like systems, the standard library modules should be in subdirectory *lib/python-2.2* for Python 2.2, *lib/python-2.3* for Python 2.3, and so on.

PYTHONPATH

A list of directories, separated by colons on Unix-like systems and by semicolons on Windows. Modules are imported from these directories. This extends the initial value for Python's sys.path variable. Modules, importing, and the sys.path variable are covered in Chapter 7.

PYTHONSTARTUP

The name of a Python source file that is automatically executed each time an interactive interpreter session starts. No such file is run if this variable is not set, or if it is set to the path of a file that is not found. The PYTHONSTARTUP file is not used when you run a Python script: it is used only when you start an interactive session.

How you set and examine environment variables depends on your operating system: shell commands, persistent startup shell files (e.g., *AUTOEXEC.BAT* on Windows), or other approaches (e.g., Start ⟶ Settings ⟶ Control Panel ⟶ System ⟶ Environment on Windows/NT, 2000, and XP). Some Python versions for Windows also look for this information in the registry, in addition to the environment. On Macintosh systems, the Python interpreter is started through the PythonInterpreter icon and configured through the EditPythonPrefs icon. See http://www.python.org/doc/current/mac/mac.html for information about Python on the Mac.

3.1.2 Command-Line Syntax and Options

The Python interpreter command-line syntax can be summarized as follows:

```
[path]python {options} [ -c command | file | - ] {arguments}
```

Here, brackets ([ ]) denote something that is optional, braces ({ }) enclose items of which 0 or more may be present, and vertical bars (|) show a choice between alternatives (with none of them also being a possibility).

options are case-sensitive short strings, starting with a hyphen, that ask *python* for a non-default behavior. Unlike most Windows programs, *python* only accepts options starting with a hyphen, not with a slash. Python consistently

3.2 Python Development Environments

The Python interpreter's built-in interactive mode is the simplest development environment for Python. It is a bit primitive, but it is lightweight, has a small footprint, and starts fast. Together with an appropriate text editor (as discussed later in this chapter) and line-editing and history facilities, it is a usable and popular development environment. However, there are a number of other development environments that you can also use.

3.2.1 IDLE

Python's Integrated DeveLopment Environment (IDLE) comes with the standard Python distribution. IDLE is a cross-platform, 100% pure Python application based on Tkinter (see Chapter 16). IDLE offers a Python shell, similar to interactive Python interpreter sessions but richer in functionality. It also includes a text editor optimized to edit Python source code, an integrated interactive debugger, and several specialized browsers/viewers.

3.2.2 Other Free Cross-Platform Python IDEs

IDLE is mature, stable, easy to use, and rich in functionality. Promising new Python IDEs that share IDLE's free and cross-platform nature are emerging. Red Hat's Source Navigator (http://sources.redhat.com/sourcenav/) supports many languages. It runs on Linux, Solaris, HPUX, and Windows. Boa Constructor (http://boa-constructor.sf.net/) is Python-only and still beta-level, but well worth trying out. Boa Constructor includes a GUI builder for the wxWindows cross-platform GUI toolkit.

3.2.3 Platform-Specific Free Python IDEs

Python is cross-platform, and this book focuses on cross-platform tools and components. However, Python also provides good platform-specific facilities, including IDEs, on many platforms it supports. For the Macintosh, MacPython includes an IDE (see http://www.python.org/doc/current/mac/mac.html). On Windows, ActivePython includes the PythonWin IDE. PythonWin is also available as a free add-on to the standard Python distribution for Windows, part of Mark Hammond's powerful win32all extensions (see http://starship.python.net/crew/mhammond).

3.2.4 Commercial Python IDEs

Several companies sell commercial Python IDEs, both cross-platform and platform-specific. You must pay for them if you use them for commercial development and, in most cases, even if you develop free software. However, they offer support contracts and rich arrays of tools. If you have funding for software tool purchases, it is worth looking at these in detail and trying out their free demos or evaluations. Most work on Linux and Windows.

Secret Labs (http://www.pythonware.com) offers a Python IDE called PythonWorks. It includes a GUI designer for Tkinter (covered in Chapter 16). Archaeopterix sells a Python IDE, Wing, notable for its powerful source-browsing and remote-debugging facilities (http://archaeopterix.com/wingide). theKompany sells a Python IDE, BlackAdder, that includes a GUI builder for the PyQt GUI toolkit (http://www.thekompany.com/products/blackadder).

ActiveState (http://www.activestate.com) has two Python IDE products. Komodo is built on top of Mozilla (http://www.mozilla.org) and includes remote debugging capabilities. Visual Python is for Windows only, and lets you use Microsoft's multi-language Visual Studio .NET IDE for Python development.

3.2.5 Free Text Editors with Python Support

3.3 Running Python Programs

Whatever tools you use to produce your Python application, you can see your application as a set of Python source files. A *script* is a file that you can run directly. A *module* is a file that you can import (as covered in Chapter 7) to provide functionality to other files or to interactive sessions. A Python file can be both a module and a script, exposing functionality when imported, but also suitable for being run directly. A useful and widespread convention is that Python files that are primarily meant to be imported as modules, when run directly, should execute self-test operations. Testing is covered in Chapter 17.

The Python interpreter automatically compiles Python source files as needed. Python source files normally have extension *.py*. Python saves the compiled bytecode file for each module in the same directory as the module's source, with the same basename and extension *.pyc* (or *.pyo* if Python is run with option -O). Python does not save the compiled bytecode form of a script when you run the script directly; rather, Python recompiles the script each time you run it. Python saves bytecode files only for modules you import. It automatically rebuilds each module's bytecode file whenever necessary, for example when you edit the module's source. Eventually, for deployment, you may package Python modules using tools covered in Chapter 26.

You can run Python code interactively, with the Python interpreter or an IDE. Normally, however, you initiate execution by running a top-level script. To run a script, you give its path as an argument to *python*, as covered earlier in this chapter. Depending on your operating system, you can invoke *python* directly, from a shell script, or in a command file. On Unix-like systems, you can make a Python script directly executable by setting the file's permission bits x and r and beginning the script with a so-called *shebang* line, which is a first line of the form:

```
#!/usr/bin/env python {options}
```

providing a path to the *python* program.

On Windows, you can associate file extensions *.py*, *.pyc*, and *.pyo* with the Python interpreter in the Windows registry. Most Python versions for Windows perform this association when installed. You can then run Python scripts with the usual Windows mechanisms, such as double-clicking on their icons. On Windows, when you run a Python script by double-clicking on the script's icon, Windows automatically closes the text-mode console associated with the script as soon as the script terminates. If you want the console to linger in order to allow the user to read the script's output on the screen, you need to ensure the script doesn't terminate too soon, for example by using the following as the script's last statement:

```
raw_input('Press Enter to terminate')
```

This is not necessary when you run the script from a pre-existing console (also known as a MS-DOS Prompt or Command Prompt window).

On Windows, you can also use extension *.pyw* and interpreter program *pythonw.exe* instead of *.py* and *python.exe*. The *w* variants run Python without a text-mode console, and thus without standard input and output. These variants are appropriate for scripts that rely on GUIs. You normally use them only when the script is fully debugged, to keep standard output and error available for information, warnings, and error messages during development.

Applications coded in other languages may embed Python, controlling the execution of Python code for their own purposes. We examine this subject further in Chapter 24.

3.4 The Jython Interpreter

The *jython* interpreter built during installation (see Chapter 2) is run similarly to the *python* program:

```
[path]jython {options} [ -j jar | -c command | file | - ] {arguments}
```

-j *jar* tells *jython* that the main script to run is *__run__.py* in the *.jar* file. Options -i, -S, and -v are the same as for *python*. --help is like *python*'s -h, and --version is like *python*'s --V. Instead of environment variables, *jython* uses a text file named *registry* in the installation directory to record properties with structured names. Property python.path, for example, is the Jython equivalent of Python's environment variable PYTHONPATH. You can also set properties with *jython* command-line options, in the form -D *name=value*.

Part II: Core Python Language and Built-ins

Chapter 4. The Python Language

This chapter is a quick guide to the Python language. To learn Python from scratch, I suggest you start with Learning Python, by Mark Lutz and David Ascher (O'Reilly). If you already know other programming languages and just want to learn the specifics of Python, this chapter is for you. I'm not trying to teach Python here, so we're going to cover a lot of ground at a pretty fast pace.

4.1 Lexical Structure

The lexical structure of a programming language is the set of basic rules that govern how you write programs in that language. It is the lowest-level syntax of the language and specifies such things as what variable names look like and what characters are used for comments. Each Python source file, like any other text file, is a sequence of characters. You can also usefully see it as a sequence of lines, tokens, or statements. These different syntactic views complement and reinforce each other. Python is very particular about program layout, especially with regard to lines and indentation, so you'll want to pay attention to this information if you are coming to Python from another language.

4.1.1 Lines and Indentation

A Python program is composed of a sequence of *logical lines*, each made up of one or more *physical lines*. Each physical line may end with a comment. A pound sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the physical line end are part of the comment, and the Python interpreter ignores them. A line containing only whitespace, possibly with a comment, is called a *blank line*, and is ignored by the interpreter. In an interactive interpreter session, you must enter an empty physical line (without any whitespace or comment) to terminate a multiline statement.

In Python, the end of a physical line marks the end of most statements. Unlike in other languages, Python statements are not normally terminated with a delimiter, such as a semicolon (;). When a statement is too long to fit on a single physical line, you can join two adjacent physical lines into a logical line by ensuring that the first physical line has no comment and ends with a backslash (\). Python also joins adjacent physical lines into one logical line if an open parenthesis ((), bracket ([), or brace ({) has not yet been closed. Triple-quoted string literals can also span physical lines. Physical lines after the first one in a logical line are known as *continuation lines*. The indentation issues covered next do not apply to continuation lines, but only to the first physical line of each logical line.

Python uses indentation to express the block structure of a program. Unlike other languages, Python does not use braces or begin/end delimiters around blocks of statements: indentation is the only way to indicate such blocks. Each logical line in a Python program is indented by the whitespace on its left. A block is a contiguous sequence of logical lines, all indented by the same amount; the block is ended by a logical line with less indentation. All statements in a block must have the same indentation, as must all clauses in a compound statement. Standard Python style is to use four spaces per indentation level. The first statement in a source file must have no indentation (i.e., it must not begin with any whitespace). Additionally, statements typed at the interactive interpreter prompt >>> (covered in Chapter 3) must have no indentation.

A tab is logically replaced by up to 8 spaces, so that the next character after the tab falls into logical column 9, 17, 25, etc. Don't mix spaces and tabs for indentation, since different tools (e.g., editors, email systems, printers) treat tabs differently. The -t and -tt options to the Python interpreter (covered in Chapter 3) ensure against inconsistent tab and space usage in Python source code. You can configure any good editor to expand tabs to spaces so that all Python source code you write contains only spaces, not tabs. You then know that all tools, including Python itself, are going to be consistent in handling the crucial matter of indentation in your source files.

4.1.2 Tokens

Python breaks each logical line into a sequence of elementary lexical components, called *tokens*. Each token corresponds to a substring of the logical line. The normal token types are identifiers, keywords, operators, delimiters, and literals, as covered in the following sections. Whitespace may be freely used between tokens to separate them. Some whitespace separation is needed between logically adjacent identifiers or keywords; otherwise, they would be parsed as a single, longer identifier. For example, printx is a single identifier—to write the keyword print followed by identifier x, you need to insert some whitespace (e.g., print x).

4.2 Data Types

The operation of a Python program hinges on the data it handles. All data values in Python are represented by objects, and each object, or value, has a type. An object's type determines what operations the object supports, or, in other words, what operations you can perform on the data value. The type also determines the object's attributes and items (if any) and whether the object can be altered. An object that can be altered is known as a mutable object, while one that cannot be altered is an immutable object. I cover object attributes and items in detail later in this chapter.

The built-in type(*obj*) accepts any object as its argument and returns the type object that represents the type of *obj*. Another built-in function, isinstance(*obj*,*type*), returns True if object *obj* is represented by type object *type*; otherwise, it returns False (built-in names True and False were introduced in Python 2.2.1; in older versions, 1 and 0 are used instead).

Python has built-in objects for fundamental data types such as numbers, strings, tuples, lists, and dictionaries, as covered in the following sections. You can also create user-defined objects, known as classes, as discussed in detail in Chapter 5.

4.2.1 Numbers

The built-in number objects in Python support integers (plain and long), floating-point numbers, and complex numbers. All numbers in Python are immutable objects, meaning that when you perform an operation on a number object, you always produce a new number object. Operations on numbers, called arithmetic operations, are covered later in this chapter.

Integer literals can be decimal, octal, or hexadecimal. A decimal literal is represented by a sequence of digits where the first digit is non-zero. An octal literal is specified with a 0 followed by a sequence of octal digits (0 to 7). To indicate a hexadecimal literal, use 0x followed by a sequence of hexadecimal digits (0 to 9 and A to F, in either upper- or lowercase). For example:

```
 1, 23, 3493                  # Decimal integers
01, 027, 06645               # Octal integers
0x1, 0x17, 0xDA5             # Hexadecimal integers
```

Any kind of integer literal may be followed by the letter L or l to denote a long integer. For instance:

```
 1L, 23L, 99999333493L        # Long decimal integers
01L, 027L, 01351033136165L   # Long octal integers
0x1L, 0x17L, 0x17486CBC75L   # Long hexadecimal integers
```

Use uppercase L here, not lowercase l, which may look like the digit 1. The difference between a long integer and a plain integer is that a long integer has no predefined size limit: it may be as large as memory allows. A plain integer takes up a few bytes of memory and has minimum and maximum values that are dictated by machine architecture. sys.maxint is the largest available plain integer, while -sys.maxint-1 is the largest negative one. On typical 32-bit machines, sys.maxint is 2147483647.

A floating-point literal is represented by a sequence of decimal digits that includes a decimal point (.), an exponent part (an e or E, optionally followed by + or -, followed by one or more digits), or both. The leading character of a floating-point literal cannot be e or E: it may be any digit or a period (.) (prior to Python 2.2, a leading 0 had to be immediately followed by a period). For example:

```
 0., 0.0, .0, 1., 1.0, 1e0, 1.e0, 1.0e0
```

A Python floating-point value corresponds to a C double and shares its limits of range and precision, typically 53 bits

4.3 Variables and Other References

A Python program accesses data values through references. A reference is a name that refers to the specific location in memory of a value (object). References take the form of variables, attributes, and items. In Python, a variable or other reference has no intrinsic type. The object to which a reference is bound at a given time does have a type, however. Any given reference may be bound to objects of different types during the execution of a program.

4.3.1 Variables

In Python, there are no declarations. The existence of a variable depends on a statement that *binds* the variable, or, in other words, that sets a name to hold a reference to some object. You can also *unbind* a variable by resetting the name so it no longer holds a reference. Assignment statements are the most common way to bind variables and other references. The del statement unbinds references.

Binding a reference that was already bound is also known as *rebinding* it. Whenever binding is mentioned in this book, rebinding is implicitly included except where it is explicitly excluded. Rebinding or unbinding a reference has no effect on the object to which the reference was bound, except that an object disappears when nothing refers to it. The automatic cleanup of objects to which there are no references is known as *garbage collection*.

You can name a variable with any identifier except the 29 that are reserved as Python's keywords (see Section 4.1.2.2 earlier in this chapter). A variable can be global or local. A global variable is an attribute of a module object (Chapter 7 covers modules). A local variable lives in a function's local namespace (see Section 4.10 later in this chapter).

4.3.1.1 Object attributes and items

The distinction between attributes and items of an object is in the syntax you use to access them. An *attribute* of an object is denoted by a reference to the object, followed by a period (.), followed by an identifier called the *attribute name* (i.e., *x*.*y* refers to the attribute of object *x* that is named *y*).

An *item* of an object is denoted by a reference to the object, followed by an expression within brackets ([ ]). The expression in brackets is called the *index* or *key* to the item, and the object is called the *container* of the item (i.e., *x*[*y*] refers to the item at key or index *y* in container object *x*).

Attributes that are callable are also known as *methods*. Python draws no strong distinction between callable and non-callable attributes, as other languages do. General rules about attributes also apply to callable attributes (methods).

4.3.1.2 Accessing nonexistent references

A common programming error is trying to access a reference that does not exist. For example, a variable may be unbound, or an attribute name or item index may not be valid for the object to which you apply it. The Python compiler, when it analyzes and compiles source code, diagnoses only syntax errors. Compilation does not diagnose semantic errors such as trying to access an unbound attribute, item, or variable. Python diagnoses semantic errors only when the errant code executes, i.e., at runtime. When an operation is a Python semantic error, attempting it raises an exception (see Chapter 6). Accessing a nonexistent variable, attribute, or item, just like any other semantic error, raises an exception.

4.4 Expressions and Operators

An expression is a phrase of code that the Python interpreter can evaluate to produce a value. The simplest expressions are literals and identifiers. You build other expressions by joining subexpressions with the operators and/or delimiters in Table 4-2. This table lists the operators in decreasing order of precedence, so operators with higher precedence are listed before those with lower precedence. Operators listed together have the same precedence. The A column lists the associativity of the operator, which can be L (left-to-right), R (right-to-left), or NA (non-associative).

In Table 4-2, *expr*, *key*, *f*, *index*, *x*, and *y* indicate any expression, while *attr* and *arg* indicate identifiers. The notation ,... indicates that commas join zero or more repetitions, except for string conversion, where one or more repetitions are allowed. A trailing comma is also allowed and innocuous in all such cases, except with string conversion, where it's forbidden.

Table 4-2. Operator precedence in expressions

Operator	**Description**	**A**
`` `expr,...` ``	String conversion	NA
`{key:expr,...}`	Dictionary creation	NA
`[expr,...]`	List creation	NA
`(expr,...)`	Tuple creation or simple parentheses	NA
`f(expr,...)`	Function call	L
`x[index:index]`	Slicing	L
`x[index]`	Indexing	L
`x.attr`	Attribute reference	L
`x**y`	Exponentiation (*x* to *y*th power)	R
`~x`	Bitwise NOT	NA
`+x`, `-x`	Unary plus and minus	NA

4.5 Numeric Operations

Python supplies the usual numeric operations, as you've just seen in Table 4-2. All numbers are immutable objects, so when you perform a numeric operation on a number object, you always produce a new number object. You can access the parts of a complex object z as read-only attributes z.real and z.imag. Trying to rebind these attributes on a complex object raises an exception.

Note that a number's optional + or - sign, and the + that joins a floating-point literal to an imaginary one to make a complex number, are not part of the literals' syntax. They are ordinary operators, subject to normal operator precedence rules (see Table 4-2). This is why, for example, -2**2 evaluates to -4: exponentiation has higher precedence than unary minus, so the whole expression parses as -(2**2), not as (-2)**2.

4.5.1 Coercion and Conversions

You can perform arithmetic operations and comparisons between any two numbers. If the operands' types differ, *coercion* applies: Python converts the operand with the smaller type to the larger type. The types, in order from smallest to largest, are integers, long integers, floating-point numbers, and complex numbers.

You can also perform an explicit conversion by passing a numeric argument to any of the built-ins: int, long, float, and complex. int and long drop their argument's fractional part, if any (e.g., int(9.8) is 9). Converting from a complex number to any other numeric type drops the imaginary part. You can also call complex with two arguments, giving real and imaginary parts.

Each built-in type can also take a string argument with the syntax of an appropriate numeric literal with two small extensions: the argument string may start with a sign and, for complex numbers, may sum or subtract real and imaginary parts. int and long can also be called with two arguments: the first one a string to convert, and the second one the radix, an integer between 2 and 36 to use as the base for the conversion (e.g., int('101',2) returns 5, the value of '101' in base 2).

4.5.2 Arithmetic Operations

If the right operand of /, //, or % is 0, Python raises a runtime exception. The // operator, introduced in Python 2.2, performs truncating division, which means it returns an integer result (converted to the same type as the wider operand) and ignores the remainder, if any. When both operands are integers, the / operator behaves like // if you are using Python 2.1 and earlier or if the switch -Qold was used on the Python command line (-Qold is the default in Python 2.2). Otherwise, / performs true division, returning a floating-point result (or a complex result, if either operand is a complex number). To have / perform true division on integer operands in Python 2.2, use the switch -Qnew on the Python command line or begin your source file with the statement:

```
from future import division
```

This ensures that operator / works without truncation on any type of operands.

To ensure that your program's behavior does not depend on the -Q switch, use // (in Python 2.2 and later) to get truncating division. When you do not want truncation, ensure that at least one operand is not an integer. For example, instead of a/b, use 1.*a/b to avoid making any assumption on the types of a and b. To check whether your program has version dependencies in its use of division, use the switch -Qwarn on the Python command line (in Python 2.2 and later) to get warnings about uses of / on integer operands.

4.6 Sequence Operations

Python supports a variety of operations that can be applied to sequence types, including strings, lists, and tuples.

4.6.1 Sequences in General

Sequences are containers with items accessible by indexing or slicing, as we'll discuss shortly. The built-in len function takes a container as an argument and returns the number of items in the container. The built-in min and max functions take one argument, a non-empty sequence (or other iterable) whose items are comparable, and they return the smallest and largest items in the sequence, respectively. You can also call min and max with multiple arguments, in which case they return the smallest and largest arguments, respectively.

4.6.1.1 Coercion and conversions

There is no implicit coercion between different sequence types except that normal strings are coerced to Unicode strings if needed. Conversion to strings is covered in detail in Chapter 9. You can call the built-in tuple and list functions with a single argument (a sequence or other iterable) to get an instance of the type you're calling, with the same items in the same order as in the argument.

4.6.1.2 Concatenation

You can concatenate sequences of the same type with the + operator. You can also multiply any sequence *S* by an integer *n* with the * operator. The result of *S***n* or *n***S* is the concatenation of *n* copies of *S*. If *n* is zero or less than zero, the result is an empty sequence of the same type as *S*.

4.6.1.3 Sequence membership

The *x* in *S* operator tests to see whether object *x* equals any item in the sequence *S*. It returns True if it does and False if it doesn't. Similarly, the *x* not in *S* operator is just like not (*x* in *S*).

4.6.1.4 Indexing a sequence

The *n*th item of a sequence *S* is denoted by an *indexing*: *S*[*n*]. Indexing in Python is zero-based (i.e., the first item in *S* is *S*[0]). If *S* has *L* items, the index *n* may be 0, 1, ... up to and including *L*-1, but no larger. *n* may also be -1, -2, ... down to and including -*L*, but no smaller. A negative *n* indicates the same item in *S* as *L*+*n* does. In other words, *S* [-1] is the last element of *S*, *S*[-2] is the next-to-last one, and so on. For example:

```
 x = [1,2,3,4]
x[1]                    # 2
x[-1]                   # 4
```

Using an index greater than or equal to *L* or less than -*L* raises an exception. Assigning to an item with an invalid index also raises an exception. You can add elements to a list, but to do so you assign to a slice, not an item, as we'll discuss shortly.

4.6.1.5 Slicing a sequence

You can denote a subsequence of *S* with a *slicing*, using the syntax *S*[*i*:*j*], where *i* and *j* are integers. *S*[*i*:*j*] is the subsequence of *S* from the *i*th item, included, to the *j*th item, excluded. Note that in Python, all ranges include the

4.7 Dictionary Operations

Python provides a variety of operations that can be applied to dictionaries. Since dictionaries are containers, the built-in len function can take a dictionary as its single argument and return the number of items (key/value pairs) in the dictionary object.

4.7.1 Dictionary Membership

In Python 2.2 and later, the *k* in *D* operator tests to see whether object *k* is one of the keys of the dictionary *D*. It returns True if it is and False if it isn't. Similarly, the *k* not in *D* operator is just like not (*k* in *D*).

4.7.2 Indexing a Dictionary

The value in a dictionary *D* that is currently associated with key *k* is denoted by an *indexing*: *D*[*k*]. Indexing with a key that is not present in the dictionary raises an exception. For example:

```
 d = { 'x':42, 'y':3.14, 'z':7 }
d['x']                           # 42
d['z']                           # 7
d['a']                           # raises exception
```

Plain assignment to a dictionary indexed with a key that is not yet in the dictionary (e.g., *D*[*newkey*]=*value*) is a valid operation that adds the key and value as a new item in the dictionary. For instance:

```
 d = { 'x':42, 'y':3.14, 'z':7 }
d['a'] = 16                      # d is now {'x':42,'y':3.14,'z':7,'a':16}
```

The del statement, in the form del *D*[*k*], removes from the dictionary the item whose key is *k*. If *k* is not a key in dictionary *D*, del *D*[*k*] raises an exception.

4.7.3 Dictionary Methods

Dictionary objects provide several methods, as shown in Table 4-4. Non-mutating methods return a result without altering the object to which they apply, while mutating methods may alter the object to which they apply. In Table 4-4 , *D* and *D1* indicate any dictionary object, *k* any valid key in *D*, and *x* any object.

Table 4-4. Dictionary object methods

Method	**Description**
Non-mutating methods	
`D.copy( )`	Returns a (shallow) copy of the dictionary
`D.has_key(k)`	Returns True if *k* is a key in *D*, otherwise returns False
	Returns a copy of the list of all items (key/value pairs) in

4.8 The print Statement

A print statement is denoted by the keyword print followed by zero or more expressions separated by commas. print is a handy, simple way to output values in text form. print outputs each expression *x* as a string that's just like the result of calling str(*x*) (covered in Chapter 8). print implicitly outputs a space between expressions, and it also implicitly outputs \n after the last expression, unless the last expression is followed by a trailing comma (,). Here are some examples of print statements:

```
 letter = 'c'
print "give me a", letter, "..."           # prints: give me a c ...
answer = 42
print "the answer is:", answer             # prints: the answer is: 42
```

The destination of print's output is the file or file-like object that is the value of the stdout attribute of the sys module (covered in Chapter 8). You can control output format more precisely by performing string formatting yourself, with the % operator or other string manipulation techniques, as covered in Chapter 9. You can also use the write or writelines methods of file objects, as covered in Chapter 10. However, print is very simple to use, and simplicity is an important advantage in the common case where all you need are the simple output strategies that print supplies.

4.9 Control Flow Statements

A program's *control flow* is the order in which the program's code executes. The control flow of a Python program is regulated by conditional statements, loops, and function calls. This section covers the if statement and for and while loops; functions are covered later in this chapter. Raising and handling exceptions also affects control flow; exceptions are covered in Chapter 6.

4.9.1 The if Statement

Often, you need to execute some statements only if some condition holds, or choose statements to execute depending on several mutually exclusive conditions. The Python compound statement if, which uses if, elif, and else clauses, lets you conditionally execute blocks of statements. Here's the syntax for the if statement:

```
if expression:
    statement(s)
elif expression:
    statement(s)
elif expression:
    statement(s)
...
else expression:
    statement(s)
```

The elif and else clauses are optional. Note that unlike some languages, Python does not have a switch statement, so you must use if, elif, and else for all conditional processing.

Here's a typical if statement:

```
if x < 0: print "x is negative"
elif x % 2: print "x is positive and odd"
else: print "x is even and non-negative"
```

When there are multiple statements in a clause (i.e., the clause controls a block of statements), the statements are placed on separate logical lines after the line containing the clause's keyword (known as the header line of the clause) and indented rightward from the header line. The block terminates when the indentation returns to that of the clause header (or further left from there). When there is just a single simple statement, as here, it can follow the : on the same logical line as the header, but it can also be placed on a separate logical line, immediately after the header line and indented rightward from it. Many Python practitioners consider the separate-line style more readable:

```
if x < 0:
    print "x is negative"
elif x % 2:
    print "x is positive and odd"
else:
    print "x is even and non-negative"
```

You can use any Python expression as the condition in an if or elif clause. When you use an expression this way, you are using it in a *Boolean context*. In a Boolean context, any value is taken as either true or false. As we discussed earlier, any non-zero number or non-empty string, tuple, list, or dictionary evaluates as true. Zero (of any numeric type), None, and empty strings, tuples, lists, and dictionaries evaluate as false. When you want to test a value *x* in a Boolean context, use the following coding style:

```
if x:
```

This is the clearest and most Pythonic form. Don't use:

```
if x is True:
if x = = True:
if bool(x):
```

4.10 Functions

Most statements in a typical Python program are organized into functions. A *function* is a group of statements that executes upon request. Python provides many built-in functions and allows programmers to define their own functions. A request to execute a function is known as a *function call*. When a function is called, it may be passed arguments that specify data upon which the function performs its computation. In Python, a function always returns a result value, either None or a value that represents the results of its computation. Functions defined within class statements are also called *methods*. Issues specific to methods are covered in Chapter 5; the general coverage of functions in this section, however, also applies to methods.

In Python, functions are objects (values) and are handled like other objects. Thus, you can pass a function as an argument in a call to another function. Similarly, a function can return another function as the result of a call. A function, just like any other object, can be bound to a variable, an item in a container, or an attribute of an object. Functions can also be keys into a dictionary. For example, if you need to quickly find a function's inverse given the function, you could define a dictionary whose keys and values are functions and then make the dictionary bidirectional (using some functions from module math, covered in Chapter 15):

```
inverse = {sin:asin, cos:acos, tan:atan, log:exp}
for f in inverse.keys(  ): inverse[inverse[f]] = f
```

The fact that functions are objects in Python is often expressed by saying that functions are first-class objects.

4.10.1 The def Statement

The def statement is the most common way to define a function. def is a single-clause compound statement with the following syntax:

```
def function-name(parameters):
    statement(s)
```

function-name is an identifier. It is a variable that gets bound (or rebound) to the function object when def executes.

parameters is an optional list of identifiers, called formal parameters or just parameters, that are used to represent values that are supplied as arguments when the function is called. In the simplest case, a function doesn't have any formal parameters, which means the function doesn't take any arguments when it is called. In this case, the function definition has empty parentheses following *function-name*.

When a function does take arguments, *parameters* contains one or more identifiers, separated by commas (,). In this case, each call to the function supplies values, known as *arguments*, that correspond to the parameters specified in the function definition. The parameters are local variables of the function, as we'll discuss later in this section, and each call to the function binds these local variables to the corresponding values that the caller supplies as arguments.

The non-empty sequence of statements, known as the *function body*, does not execute when the def statement executes. Rather, the function body executes later, each time the function is called. The function body can contain zero or more occurrences of the return statement, as we'll discuss shortly.

Here's an example of a simple function that returns a value that is double the value passed to it:

```
def double(x):
    return x*2
```

4.10.2 Parameters

Chapter 5. Object-Oriented Python

Python is an object-oriented programming language. Unlike some other object-oriented languages, Python doesn't force you to use the object-oriented paradigm exclusively. Python also supports procedural programming with modules and functions, so you can select the most suitable programming paradigm for each part of your program. Generally, the object-oriented paradigm is suitable when you want to group state (data) and behavior (code) together in handy packets of functionality. It's also useful when you want to use some of Python's object-oriented mechanisms covered in this chapter, such as inheritance or special methods. The procedural paradigm, based on modules and functions, tends to be simpler and is more suitable when you don't need any of the benefits of object-oriented programming. With Python, you often mix and match the two paradigms.

Python 2.2 and 2.3 are in transition between two slightly different object models. This chapter starts by describing the classic object model, which was the only one available in Python 2.1 and earlier and is still the default model in Python 2.2 and 2.3. The chapter then covers the small differences that define the powerful new-style object model and discusses how to use the new-style object model with Python 2.2 and 2.3. Because the new-style object model builds on the classic one, you'll need to understand the classic model before you can learn about the new model. Finally, the chapter covers special methods for both the classic and new-style object models, as well as metaclasses for Python 2.2 and later.

The new-style object model will become the default in a future version of Python. Even though the classic object model is still the default, I suggest you use the new-style object model when programming with Python 2.2 and later. Its advantages over the classic object model, while small, are measurable, and there are practically no compensating disadvantages. Therefore, it's simpler just to stick to the new-style object model, rather than try to decide which model to use each time you code a new class.

5.1 Classic Classes and Instances

A classic class is a Python object with several characteristics:

- You can call a class object as if it were a function. The call creates another object, known as an *instance* of the class, that knows what class it belongs to.
- A class has arbitrarily named attributes that you can bind and reference.
- The values of class attributes can be data objects or function objects.
- Class attributes bound to functions are known as *methods* of the class.
- A method can have a special Python-defined name with two leading and two trailing underscores. Python invokes such *special methods*, if they are present, when various kinds of operations take place on class instances.
- A class can *inherit* from other classes, meaning it can delegate to other class objects the lookup of attributes that are not found in the class itself.

An instance of a class is a Python object with arbitrarily named attributes that you can bind and reference. An instance object implicitly delegates to its class the lookup of attributes not found in the instance itself. The class, in turn, may delegate the lookup to the classes from which it inherits, if any.

In Python, classes are objects (values), and are handled like other objects. Thus, you can pass a class as an argument in a call to a function. Similarly, a function can return a class as the result of a call. A class, just like any other object, can be bound to a variable (local or global), an item in a container, or an attribute of an object. Classes can also be keys into a dictionary. The fact that classes are objects in Python is often expressed by saying that classes are first-class objects.

5.1.1 The class Statement

The class statement is the most common way to create a class object. class is a single-clause compound statement with the following syntax:

```
class classname[(base-classes)]:
    statement(s)
```

classname is an identifier. It is a variable that gets bound (or rebound) to the class object after the class statement finishes executing.

5.2 New-Style Classes and Instances

Most of what I have covered so far in this chapter also holds for the new-style object model introduced in Python 2.2. New-style classes and instances are first-class objects just like classic ones, both can have arbitrary attributes, you call a class to create an instance of the class, and so on. In this section, I'm going to cover the few differences between the new-style and classic object models.

In Python 2.2 and 2.3, a class is new-style if it inherits from built-in type object directly or indirectly (i.e., if it subclasses any built-in type, such as list, dict, file, object, and so on). In Python 2.1 and earlier, a class cannot inherit from a built-in type, and built-in type object does not exist. In Section 5.4 later in this chapter, I cover other ways to make a class new-style, ways that you can use in Python 2.2 or later whether a class has superclasses or not.

As I said at the beginning of this chapter, I suggest you get into the habit of using new-style classes when you program in Python 2.2 or later. The new-style object model has small but measurable advantages, and there are practically no compensating disadvantages. It's simpler just to stick to the new-style object model, rather than try to decide which model to use each time you code a new class.

5.2.1 The Built-in object Type

As of Python 2.2, the built-in object type is the ancestor of all built-in types and new-style classes. The object type defines some special methods (as documented in Section 5.3 later in this chapter) that implement the default semantics of objects:

_ _new_ _ , _ _init_ _

You can create a direct instance of object, and such creation implicitly uses the static method _ _new_ _ of type object to create the new instance, and then uses the new instance's _ _init_ _ method to initialize the new instance. object._ _init_ _ ignores its arguments and performs no operation whatsoever, so you can pass arbitrary arguments to type object when you call it to create an instance of it: all such arguments will be ignored.

_ _delattr_ _ , _ _getattribute_ _, _ _setattr_ _

By default, an object handles attribute references as covered earlier in this chapter, using these methods of object.

_ _hash_ _ , _ _repr_ _, _ _str_ _

An object can be passed to functions hash and repr and to type str.

A subclass of object may override any of these methods and/or add others.

5.2.2 Class-Level Methods

The new-style object model allows two kinds of class-level methods that do not exist in the classic object model: static methods and class methods. Class-level methods exist only in Python 2.2 and later, but in these versions you can also have such methods in classic classes. This is the only feature of the new-style object model that is also fully functional with classic classes in Python 2.2 and later.

5.2.2.1 Static methods

A *static method* is a method that you can call on a class, or on any instance of the class, without the special behavior and constraints of ordinary methods, bound and unbound, on the first argument. A static method may have any

5.3 Special Methods

A class may define or inherit special methods (i.e., methods whose names begin and end with double underscores). Each special method relates to a specific operation. Python implicitly invokes a special method whenever you perform the related operation on an instance object. In most cases, the method's return value is the operation's result, and attempting an operation when its related method is not present raises an exception. Throughout this section, I will point out the cases in which these general rules do not apply. In the following, *x* is the instance of class *C* on which you perform the operation, and *y* is the other operand, if any. The formal argument self of each method also refers to instance object *x*.

5.3.1 General-Purpose Special Methods

Some special methods relate to general-purpose operations. A class that defines or inherits these methods allows its instances to control such operations. These operations can be divided into the following categories:
Initialization and finalization

An instance can control its initialization (a frequent need) via special method _ _init_ _, and/or its finalization (a rare need) via _ _del_ _.
Representation as string

An instance can control how Python represents it as a string via special methods _ _repr_ _, _ _str_ _, and _ _unicode_ _.
Comparison, hashing, and use in a Boolean context

An instance can control how it compares with other objects (methods _ _lt_ _ and _ _cmp_ _), how dictionaries use it as a key (_ _hash_ _), and whether it evaluates to true or false in Boolean contexts (_ _nonzero_ _).
Attribute reference, binding, and unbinding

An instance can control access to its attributes (reference, binding, unbinding) by defining special methods _ _getattribute_ _, _ _getattr_ _, _ _setattr_ _, and _ _delattr_ _.
Callable instances

An instance is callable, just like a function object, if it has the special method _ _call_ _.

The rest of this section documents the general-purpose special methods.

_ _call_ _

```
_ _call_ _(self[,args...])
```

When you call *x*(*[args...]*), Python translates the operation into a call to *x*._ _call_ _(*[args...]*). The formal arguments for the call operation are the same as for the _ _call_ _ method, minus the first argument. The first argument, conventionally called self, refers to *x*, and Python supplies it implicitly and automatically, just as in any other call to a bound method.

_ _cmp_ _

```
_ _cmp_ _(self,other)
```

Any comparison, when its specific special method (_ _lt_ _, _ _gt_ _, etc.) is absent or returns NotImplemented,

5.4 Metaclasses

Any object, even a class object, has a type. In Python, types and classes are also first-class objects. The type of a class object is also known as the class's *metaclass*.[1] An object's behavior is determined largely by the type of the object. This also holds for classes: a class's behavior is determined largely by the class's metaclass. Metaclasses are an advanced subject, and you may want to skip the rest of this chapter on first reading. However, fully grasping metaclasses can help you obtain a deeper understanding of Python, and sometimes it can even be useful to define your own custom metaclasses.

[1] Strictly speaking, the type of a class *C* could be said to be the metaclass only of instances of *C*, rather than of *C* itself, but this exceedingly subtle terminological distinction is rarely, if ever, observed in practice.

The distinction between classic and new-style classes relies on the fact that each class's behavior is determined by its metaclass. In other words, the reason classic classes behave differently from new-style classes is that classic and new-style classes are object of different types (metaclasses):

```
 class Classic: pass
class Newstyle(object): pass
print type(Classic)                 # prints: <type 'class'>
print type(Newstyle)                # prints: <type 'type'>
```

The type of Classic is object types.ClassType from standard module types, while the type of Newstyle is built-in object type. type is also the metaclass of all Python built-in types, including itself (i.e., print type(type) also prints <type 'type'>).

5.4.1 How Python Determines a Class's Metaclass

To execute a class statement, Python first collects the base classes into a tuple *t* (an empty one, if there are no base classes) and executes the class body in a temporary dictionary *d*. Then, Python determines the metaclass *M* to use for the new class object *C* created by the class statement.

When '_ _metaclass_ _' is a key in *d*, *M* is *d*['_ _metaclass_ _']. Thus, you can explicitly control class *C*'s metaclass by binding the attribute _ _metaclass_ _ in *C*'s class body. Otherwise, when *t* is non-empty (i.e., when *C* has one or more base classes), *M* is type(*t*[0]), the metaclass of *C*'s first base class. This is why inheriting from object indicates that *C* is a new-style class. Since type(object) is type, a class *C* that inherits from object (or some other built-in type) gets the same metaclass as object (i.e., type(*C*), *C*'s metaclass, is also type) Thus, being a new-style class is synonymous with having type as the metaclass.

When *C* has no base classes, but the current module has a global variable named _ _metaclass_ _, *M* is the value of that global variable. This lets you make classes without base classes default to new-style classes, rather than classic classes, throughout a module. Just place the following statement toward the start of the module body:

```
 _ _metaclass_ = type
```

Failing all of these, in Python 2.2 and 2.3, *M* defaults to types.ClassType. This last default of defaults clause is why classes without base classes are classic classes by default, when _ _metaclass_ _ is not bound in the class body or as a global variable of the module.

5.4.2 How a Metaclass Creates a Class

Having determined *M*, Python calls *M* with three arguments: the class name (a string), the tuple of base classes *t*, and

Chapter 6. Exceptions

Python uses exceptions to communicate errors and anomalies. An *exception* is an object that indicates an error or anomalous condition. When Python detects an error, it raises an exception; that is, it signals the occurrence of an anomalous condition by passing an exception object to the exception-propagation mechanism. Your code can also explicitly raise an exception by executing a raise statement.

Handling an exception means receiving the exception object from the propagation mechanism and performing whatever actions are needed to deal with the anomalous situation. If a program does not handle an exception, it terminates with an error traceback message. However, a program can handle exceptions and keep running despite errors or other abnormal conditions.

Python also uses exceptions to indicate some special situations that are not errors, and are not even abnormal occurrences. For example, as covered in Chapter 4, an iterator's next method raises the exception StopIteration when the iterator has no more items. This is not an error, and it is not even an anomalous condition, since most iterators run out of items eventually.

6.1 The try Statement

The try statement provides Python's exception-handling mechanism. It is a compound statement that can take one of two different forms:

- A try clause followed by one or more except clauses
- A try clause followed by exactly one finally clause

6.1.1 try/except

Here's the syntax for the try/except form of the try statement:

```
try:
    statement(s)
except [expression [, target]]:
    statement(s)
[else:
    statement(s)]
```

This form of the try statement has one or more except clauses, as well as an optional else clause.

The body of each except clause is known as an *exception handler*. The code executes if the *expression* in the except clause matches an exception object that propagates from the try clause. *expression* is an optional class or tuple of classes that matches any exception object of one of the listed classes or any of their subclasses. The optional *target* is an identifier that names a variable that Python binds to the exception object just before the exception handler executes. A handler can also obtain the current exception object by calling the exc_info function of module sys (covered in Chapter 8).

Here is an example of the try/except form of the try statement:

```
try: 1/0
except ZeroDivisionError: print "caught divide-by-0 attempt"
```

If a try statement has several except clauses, the exception propagation mechanism tests the except clauses in order: the first except clause whose expression matches the exception object is used as the handler. Thus, you must always list handlers for specific cases before you list handlers for more general cases. If you list a general case first, the more specific except clauses that follow will never enter the picture.

The last except clause may lack an expression. This clause handles any exception that reaches it during propagation. Such unconditional handling is a rare need, but it does occur, generally in wrapper functions that must perform some extra task before reraising an exception, as we'll discuss later in the chapter.

Note that exception propagation terminates when it finds a handler whose expression matches the exception object. Thus, if a try statement is nested in the try clause of another try statement, a handler established by the inner try is reached first during propagation, and therefore is the one that handles the exception, if it matches the expression. For example:

6.2 Exception Propagation

When an exception is raised, the exception-propagation mechanism takes control. The normal control flow of the program stops, and Python looks for a suitable exception handler. Python's try statement establishes exception handlers via its except clauses. The handlers deal with exceptions raised in the body of the try clause, as well as exceptions that propagate from any of the functions called by that code, directly or indirectly. If an exception is raised within a try clause that has an applicable except handler, the try clause terminates and the handler executes. When the handler finishes, execution continues with the statement after the try statement.

If the statement raising the exception is not within a try clause that has an applicable handler, the function containing the statement terminates, and the exception propagates upward to the statement that called the function. If the call to the terminated function is within a try clause that has an applicable handler, that try clause terminates, and the handler executes. Otherwise, the function containing the call terminates, and the propagation process repeats, unwinding the stack of function calls until an applicable handler is found.

If Python cannot find such a handler, by default the program prints an error message to the standard error stream (the file sys.stderr). The error message includes a traceback that gives details about functions terminated during propagation. You can change Python's default error-reporting behavior by setting sys.excepthook (covered in Chapter 8). After error reporting, Python goes back to the interactive session, if any, or terminates if no interactive session is active. When the exception class is SystemExit, termination is silent and includes the interactive session, if any.

Here are some functions that we can use to see exception propagation at work.

```
def f(  ):
    print "in f, before 1/0"
    1/0                           # raises a ZeroDivisionError exception
    print "in f, after 1/0"

def g(  ):
    print "in g, before f(  )"
    f(  )
    print "in g, after f(  )"

def h(  ):
    print "in h, before g(  )"
    try:
        g(  )
        print "in h, after g(  )"
    except ZeroDivisionError:
        print "ZD exception caught"
    print "function h ends"
```

Calling the h function has the following results:

```
>>> h(  )
in h, before g(  )
in g, before f(  )
in f, before 1/0
ZD exception caught
function h ends
```

Function h establishes a try statement and calls function g within the try clause. g, in turn, calls f, which performs a division by 0, raising an exception of class ZeroDivisionError. The exception propagates all the way back to the except clause in h. Functions f and g terminate during the exception propagation phase, which is why neither of their "after" messages is printed. The execution of h's try clause also terminates during the exception propagation phase, so its "after" message isn't printed either. Execution continues after the handler, at the end of h's try/except block.

6.3 The raise Statement

You can use the raise statement to raise an exception explicitly. raise is a simple statement with the following syntax:

```
raise [expression1[, expression2]]
```

Only an exception handler (or a function that a handler calls, directly or indirectly) can use raise without any expressions. A plain raise statement reraises the same exception object that the handler received. The handler terminates, and the exception propagation mechanism keeps searching for other applicable handlers. Using a raise without expressions is useful when a handler discovers that it is unable to handle an exception it receives, so the exception should keep propagating.

When only *expression1* is present, it can be an instance object or a class object. In this case, if *expression1* is an instance object, Python raises that instance. When *expression1* is a class object, raise instantiates the class without arguments and raises the resulting instance. When both expressions are present, *expression1* must be a class object. raise instantiates the class, with *expression2* as the argument (or multiple arguments if *expression2* is a tuple), and raises the resulting instance.

Here's an example of a typical use of the raise statement:

```
def crossProduct(seq1, seq2):
    if not seq1 or not seq2:
        raise ValueError, "Sequence arguments must be non-empty"
    return [ (x1, x2) for x1 in seq1 for x2 in seq2 ]
```

The crossProduct function returns a list of all pairs with one item from each of its sequence arguments, but first it tests both arguments. If either argument is empty, the function raises ValueError, rather than just returning an empty list as the list comprehension would normally do. Note that there is no need for crossProduct to test if seq1 and seq2 are iterable: if either isn't, the list comprehension itself will raise the appropriate exception, presumably a TypeError. Once an exception is raised, be it by Python itself or with an explicit raise statement in your code, it's up to the caller to either handle it (with a suitable try/except statement) or let it propagate further up the call stack.

Use the raise statement only to raise additional exceptions for cases that would normally be okay but your specifications define to be errors. Do not use raise to duplicate the error checking and diagnostics Python already and implicitly does on your behalf.

6.4 Exception Objects

Exceptions are instances of subclasses of the built-in Exception class. For backward compatibility, Python also lets you use strings, or instances of any class, as exception objects, but such usage risks future incompatibility and gives no benefits. An instance of any subclass of Exception has an attribute args, the tuple of arguments used to create the instance. args holds error-specific information, usable for diagnostic or recovery purposes.

6.4.1 The Hierarchy of Standard Exceptions

All exceptions that Python itself raises are instances of subclasses of Exception. The inheritance structure of exception classes is important, as it determines which except clauses handle which exceptions.

The SystemExit class inherits directly from Exception. Instances of SystemExit are normally raised by the exit function in module sys (covered in Chapter 8).

Other standard exceptions derive from StandardError, a direct subclass of Exception. Three subclasses of StandardError, like StandardError itself and Exception, are never instantiated directly. Their purpose is to make it easier for you to specify except clauses that handle a broad range of related errors. These subclasses are:
ArithmeticError

The base class for exceptions due to arithmetic errors (i.e., OverflowError, ZeroDivisionError, FloatingPointError)
LookupError

The base class for exceptions that a container raises when it receives an invalid key or index (i.e., IndexError, KeyError)
EnvironmentError

The base class for exceptions due to external causes (i.e., IOError, OSError, WindowsError)

6.4.2 Standard Exception Classes

Common runtime errors raise exceptions of the following classes:
AssertionError

An assert statement failed.
AttributeError

An attribute reference or assignment failed.
FloatingPointError

A floating-point operation failed. Derived from ArithmeticError.
IOError

An I/O operation failed (e.g., the disk is full, a file was not found, or needed permissions were missing). Derived from EnvironmentError.
ImportError

An import statement (covered in Chapter 7) cannot find the module to import or cannot find a name specifically requested from the module.
IndentationError

6.5 Custom Exception Classes

You can subclass any of the standard exception classes in order to define your own exception class. Typically, such a subclass adds nothing more than a docstring:

```
class InvalidAttribute(AttributeError):
    "Used to indicate attributes that could never be valid"
```

Given the semantics of try/except, raising a custom exception class such as InvalidAttribute is almost the same as raising its standard exception superclass, AttributeError. Any except clause able to handle AttributeError can handle InvalidAttribute just as well. In addition, client code that knows specifically about your InvalidAttribute custom exception class can handle it specifically, without having to handle all other cases of AttributeError if it is not prepared for those. For example:

```
class SomeFunkyClass(object):
    "much hypothetical functionality snipped"
    def _ _getattr_ _(self, name):
        "this _ _getattr_ _ only clarifies the kind of attribute error"
        if name.startswith('_'):
            raise InvalidAttribute, "Unknown private attribute "+name
        else:
            raise AttributeError, "Unknown attribute "+name
```

Now client code can be more selective in its handlers. For example:

```
s = SomeFunkyClass(  )
try:
    value = getattr(s, thename)
except InvalidAttribute, err
    warnings.warn(str(err))
    value = None
# other cases of AttributeError just propagate, as they're unexpected
```

A special case of custom exception class that you may sometimes find useful is one that wraps another exception and adds further information. To gather information about a pending exception, you can use the exc_info function from module sys (covered in Chapter 8). Given this, your custom exception class could be defined as follows:

```
import sys
class CustomException(Exception):
    "Wrap arbitrary pending exception, if any, in addition to other info"
    def _ _init_ _(self, *args):
        Exception._ _init_ _(self, *args)
        self.wrapped_exc = sys.exc_info(  )
```

You would then typically use this class in a wrapper function such as:

```
def call_wrapped(callable, *args, **kwds):
    try: return callable(*args, **kwds)
    except: raise CustomException, "Wrapped function propagated exception"
```

6.6 Error-Checking Strategies

Most programming languages that support exceptions are geared to raise exceptions only in very rare cases. Python's emphasis is different. In Python, exceptions are considered appropriate whenever they make a program simpler and more robust. A common idiom in other languages, sometimes known as "look before you leap" (LBYL), is to check in advance, before attempting an operation, for all circumstances that might make the operation invalid. This is not ideal, for several reasons:

- The checks may diminish the readability and clarity of the common, mainstream cases where everything is okay.
- The work needed for checking may duplicate a substantial part of the work done in the operation itself.
- The programmer might easily err by omitting some needed check.
- The situation might change between the moment the checks are performed and the moment the operation is attempted.

The preferred idiom in Python is generally to attempt the operation in a try clause and handle the exceptions that may result in except clauses. This idiom is known as "it's easier to ask forgiveness than permission" (EAFP), a motto widely credited to Admiral Grace Murray Hopper, co-inventor of COBOL, and shares none of the defects of "look before you leap." Here is a function written using the LBYL idiom:

```
def safe_divide_1(x, y):
    if y=  =0:
        print "Divide-by-0 attempt detected"
        return None
    else:
        return x/y
```

With LBYL, the checks come first, and the mainstream case is somewhat hidden at the end of the function.

Here is the equivalent function written using the EAFP idiom:

```
def safe_divide_2(x, y):
    try:
        return x/y
    except ZeroDivisionError:
        print "Divide-by-0 attempt detected"
        return None
```

With EAFP, the mainstream case is up front in a try clause, and the anomalies are handled in an except clause.

EAFP is most often the preferable error-handling strategy, but it is not a panacea. In particular, you must be careful not to cast too wide a net, catching errors that you did not expect and therefore did not mean to catch. The following is a typical case of such a risk (built-in function getattr is covered in Chapter 8):

```
def trycalling(obj, attrib, default, *args, **kwds):
    try: return getattr(obj, attrib)(*args, **kwds)
```

Chapter 7. Modules

A typical Python program is made up of several source files. Each source file corresponds to a *module*, which packages program code and data for reuse. Modules are normally independent of each other so that other programs can reuse the specific modules they need. A module explicitly establishes dependencies upon another module by using import or from statements. In some other programming languages, global variables can provide a hidden conduit for coupling between modules. In Python, however, global variables are not global to all modules, but instead such variables are attributes of a single module object. Thus, Python modules communicate in explicit and maintainable ways.

Python also supports *extensions*, which are components written in other languages, such as C, C++, or Java, for use with Python. Extensions are seen as modules by the Python code that uses them (called client code). From the client code viewpoint, it does not matter whether a module is 100% pure Python or an extension. You can always start by coding a module in Python. Later, if you need better performance, you can recode some modules in a lower-level language without changing the client code that uses the modules. Chapter 24 and Chapter 25 discuss writing extensions in C and Java.

This chapter discusses module creation and loading. It also covers grouping modules into *packages*, which are modules that contain other modules, forming a hierarchical, tree-like structure. Finally, the chapter discusses using Python's distribution utilities (distutils) to prepare packages and modules for distribution and to install distributed packages and modules.

7.1 Module Objects

A module is a Python object with arbitrarily named attributes that you can bind and reference. The Python code for a module named *aname* normally resides in a file named *aname.py*, as covered in Section 7.2 later in this chapter.

In Python, modules are objects (values) and are handled like other objects. Thus, you can pass a module as an argument in a call to a function. Similarly, a function can return a module as the result of a call. A module, just like any other object, can be bound to a variable, an item in a container, or an attribute of an object. For example, the sys.modules dictionary, covered later in this chapter, holds module objects as its values.

7.1.1 The import Statement

You can use any Python source file as a module by executing an import statement in some other code. import has the following syntax:

```
 import modname [as varname][,...]
```

The import keyword is followed by one or more module specifiers, separated by commas. In the simplest and most common case, *modname* is an identifier, the name of a variable that Python binds to the module object when the import statement finishes. In this case, Python looks for the module of the same name to satisfy the import request. For example:

```
 import MyModule
```

looks for the module named MyModule and binds the variable named MyModule in the current scope to the module object. *modname* can also be a sequence of identifiers separated by dots (.) that names a module in a package, as covered in later in this chapter.

When as *varname* is part of an import statement, Python binds the variable named *varname* to the module object, but the module name that Python looks for is *modname*. For example:

```
 import MyModule as Alias
```

looks for the module named MyModule and binds the variable named Alias in the current scope to the module object. *varname* is always a simple identifier.

7.1.1.1 Module body

The body of a module is the sequence of statements in the module's source file. There is no special syntax required to indicate that a source file is a module; any valid source file can be used as a module. A module's body executes immediately the first time the module is imported in a given run of a program. During execution of the body, the module object already exists and an entry in sys.modules is already bound to the module object.

7.1.1.2 Attributes of module objects

An import statement creates a new namespace that contains all the attributes of the module. To access an attribute in this namespace, use the name of the module object as a prefix:

```
 import MyModule
a = MyModule.f(  )
```

or:

```
 import MyModule as Alias
```

7.2 Module Loading

Module-loading operations rely on attributes of the built-in sys module (covered in Chapter 8). The module-loading process described here is carried out by built-in function __import__. Your code can call __import__ directly, with the module name string as an argument. __import__ returns the module object or raises ImportError if the import fails.

To import a module named *M*, __import__ first checks dictionary sys.modules, using string *M* as the key. When key *M* is in the dictionary, __import__ returns the corresponding value as the requested module object. Otherwise, __import__ binds sys.modules[*M*] to a new empty module object with a __name__ of *M*, then looks for the right way to initialize (load) the module, as covered in Section 7.2.2 later in this section.

Thanks to this mechanism, the loading operation takes place only the first time a module is imported in a given run of the program. When a module is imported again, the module is not reloaded, since __import__ finds and returns the module's entry in sys.modules. Thus, all imports of a module after the first one are extremely fast because they're just dictionary lookups.

7.2.1 Built-in Modules

When a module is loaded, __import__ first checks whether the module is built-in. Built-in modules are listed in tuple sys.builtin_module_names, but rebinding that tuple does not affect module loading. A built-in module, like any other Python extension, is initialized by calling the module's initialization function. The search for built-in modules also finds frozen modules and modules in platform-specific locations (e.g., resources on the Mac, the Registry in Windows).

7.2.2 Searching the Filesystem for a Module

If module *M* is not built-in or frozen, __import__ looks for *M*'s code as a file on the filesystem. __import__ looks in the directories whose names are the items of list sys.path, in order. sys.path is initialized at program startup, using environment variable PYTHONPATH (covered in Chapter 3) if present. The first item in sys.path is always the directory from which the main program (script) is loaded. An empty string in sys.path indicates the current directory.

Your code can mutate or rebind sys.path, and such changes affect what directories __import__ searches to load modules. Changing sys.path does not affect modules that are already loaded (and thus already listed in sys.modules) when sys.path is changed.

If a text file with extension *.pth* is found in the PYTHONHOME directory at startup, its contents are added to sys.path, one item per line. *.pth* files can also contain blank lines and comment lines starting with the character #, as Python ignores any such lines. *.pth* files can also contain import statements, which Python executes, but no other kinds of statements.

When looking for the file for module *M* in each directory along sys.path, Python considers the following extensions in the order listed:

1. *.pyd* and *.dll* (Windows) or *.so* (most Unix-like platforms), which indicate Python extension modules. (Some Unix dialects use different extensions; e.g., *.sl* is the extension used on HP-UX.)

2.

7.3 Packages

A *package* is a module that contains other modules. Modules in a package may be subpackages, resulting in a hierarchical tree-like structure. A package named *P* resides in a subdirectory, also called *P*, of some directory in sys.path. The module body of *P* is in the file *P/_ _init_ _.py*. You must have a file named *P/_ _init_ _.py*, even if it's empty (representing an empty module body), in order to indicate to Python that directory *P* is indeed a package. Other *.py* files in directory *P* are the modules of package *P*. Subdirectories of *P* containing *_ _init_ _.py* files are subpackages of *P*. Nesting can continue to any depth.

You can import a module named *M* in package *P* as *P*.*M*. More dots let you navigate a hierarchical package structure. A package is always loaded before a module in the package is loaded. If you use the syntax import *P*.*M*, variable *P* is bound to the module object of package *P*, and attribute *M* of object *P* is bound to module *P*.*M*. If you use the syntax import *P*.*M* as *V*, variable *V* is bound directly to module *P*.*M*.

Using from *P* import *M* to import a specific module *M* from package *P* is fully acceptable programming practice. In other words, the from statement is specifically okay in this case.

A module *M* in a package *P* can import any other module *X* of *P* with the statement import *X*. Python searches the module's own package directory before searching the directories in sys.path. However, this applies only to sibling modules, not to ancestors or other more-complicated relationships. The simplest, cleanest way to share objects (such as functions or constants) among modules in a package *P* is to group the shared objects in a file named *P/Common.py*. Then you can import Common from every module in the package that needs to access the objects, and then refer to the objects as Common.*f*, Common.*K*, and so on.

7.4 The Distribution Utilities (distutils)

Python modules, extensions, and applications can be packaged and distributed in several forms:

Compressed archive files

Generally *.zip* for Windows and *.tar.gz* or *.tgz* for Unix-based systems, but both forms are portable

Self-unpacking or self-installing executables

Normally *.exe* for Windows

Platform-specific installers

For example, *.msi* on Windows, *.rpm* and *.srpm* on Linux, and *.deb* on Debian GNU/Linux

When you distribute a package as a self-installing executable or platform-specific installer, a user can then install the package simply by running the installer. How to run such an installer program depends on the platform, but it no longer matters what language the program was written in.

When you distribute a package as an archive file or as an executable that unpacks but does not install itself, it does matter that the package was coded in Python. In this case, the user must first unpack the archive file into some appropriate directory, say *C:\Temp\MyPack* on a Windows machine or *~/MyPack* on a Unix-like machine. Among the extracted files there should be a script, conventionally named *setup.py*, that uses the Python facility known as the *distribution utilities* (package distutils). The distributed package is then almost as easy to install as a self-installing executable would be. The user opens a command-prompt window and changes to the directory into which the archive is unpacked. Then the user runs, for example:

```
C:\Temp\MyPack> python setup.py install
```

The *setup.py* script, run with this install command, installs the package as a part of the user's Python installation, according to the options specified in the setup script by the package's author. distutils, by default, provides tracing information when the user runs *setup.py*. Option --quiet, placed right before the install command, hides most details (the user still sees error messages, if any). The following command:

```
C:\> python setup.py --help
```

gives help on distutils.

When you are installing a package prepared with distutils, you can, if you wish, exert detailed control over how distutils performs installations. You can record installation options in a text file with extension *.cfg*, called a config file, so that distutils applies your favorite installation options by default. Such customization can be done on a systemwide basis, for a single user, or even for a single package installation. For example, if you want an installation with minimal amounts of output to be your systemwide default, create the following text file named *pydistutils.cfg*:

```
 [global]
quiet=1
```

Place this file in the same directory in which the distutils package resides. On a typical Python 2.2 installation on Windows, for example, the file is *C:\Python22\Lib\distutils\pydistutils.cfg*. Chapter 26 provides more information on using distutils to prepare Python modules, packages, extensions, and applications for distribution.

Chapter 8. Core Built-ins

The term built-in has more than one meaning in Python. In most contexts, a built-in is any object directly accessible to a Python program without an import statement. Chapter 7 showed the mechanism that Python uses to allow this direct access. Built-in types in Python include numbers, sequences, dictionaries, functions (covered in Chapter 4), classes (covered in Chapter 5), the standard exception classes (covered in Chapter 6), and modules (covered in Chapter 7). The built-in file object is covered in Chapter 10, and other built-in types covered in Chapter 13 are intrinsic to Python's internal operation. This chapter provides additional coverage of the core built-in types, and it also covers the built-in functions available in module __builtin__.

As I mentioned in Chapter 7, some modules are called built-in because they are an integral part of the Python standard library, even though it takes an import statement to access them. Built-in modules are distinct from separate, optional add-on modules, also called Python extensions. This chapter documents the following core built-in modules: sys, getopt, copy, bisect, UserList, UserDict, and UserString. Chapter 9 covers some string-related core built-in modules, while Parts III and IV of the book cover many other useful built-in modules.

8.1 Built-in Types

This section documents Python's core built-in types, like int, float, and dict. Note that prior to Python 2.2, these names referred to factory functions for creating objects of these types. As of Python 2.2, however, they refer to actual type objects. Since you can call type objects just as if they were functions, this change does not break existing programs.

classmethod *Python 2.2 and later*

`classmethod(function)`

Creates and returns a class method object. In practice, you call this built-in type only within a class body. See Section 5.2.2.2.

complex

`complex(real,imag=0)`

Converts any number, or a suitable string, to a complex number. *imag* may be present only when *real* is a number, and is the imaginary part of the resulting complex number.

dict *Python 2.2 and later*

`dict(x={  })`

Returns a new dictionary object with the same items as argument *x*. When *x* is a dictionary, dict(*x*) returns a copy of *x*, like *x*.copy( ) does. Alternatively, *x* can be a sequence of pairs, that is, a sequence whose items are sequences with two items each. In this case, dict(*x*) returns a dictionary whose keys are the first items of each pair in *x*, while the corresponding values are the corresponding second items. In other words, when *x* is a sequence, c=dict(*x*) has the same effect as the following:

```
 c = {  }
for key, value in x: c[key] = value
```

file, open

```
file(path,mode='r',bufsize=-1)
open(filename,mode='r',bufsize
=-1)
```

Opens or creates a file and returns a new file object. In Python 2.2 and later, open is a synonym for the built-in type file. In Python 2.1 and earlier, open was a built-in function and file was not a built-in name at all. See Section 10.3.

float

`float(x)`

Converts any number, or a suitable string, to a floating-point number.

int

8.2 Built-in Functions

This section documents the Python functions available in module _ _builtin_ _ in alphabetical order. Note that the names of these built-ins are not reserved words. Thus, your program can bind for its own purposes, in local or global scope, an identifier that has the same name as a built-in function. Names bound in local or global scope have priority over names bound in built-in scope, so local and global names hide built-in ones. You can also rebind names in built-in scope, as covered in Chapter 7. You should avoid hiding built-ins that your code might need.

_ _import_ _

```
_ _import_ _(module_name[,
globals[,locals[,fromlist]]])
```

Loads the module named by string *module_name* and returns the resulting module object. *globals*, which defaults to the result of globals( ), and *locals*, which defaults to the result of locals( ) (both covered in this section), are dictionaries that _ _import_ _ treats as read-only and uses only to get context for package-relative imports, covered in Section 7.3. *fromlist* defaults to an empty list, but can be a list of strings that name the module attributes to be imported in a from statement. See Section 7.2 for more details on module loading.

In practice, when you call _ _import_ _, you generally pass only the first argument, except in the rare and dubious case in which you use _ _import_ _ for a package-relative import. When you replace the built-in _ _import_ _ function with your own in order to provide special import functionality, you may have to take *globals*, *locals*, and *fromlist* into account.

abs

```
abs(x)
```

Returns the absolute value of number *x*. When *x* is complex, abs returns the square root of *x*.imag**2+*x*.real**2. Otherwise, abs returns -*x* if *x* is less than 0, or *x* if *x* is greater than or equal to 0. See also _ _abs_ _ in Chapter 5.

apply

```
apply(func,args=(  ),keywords={
})
```

Calls a function (or other callable object) and returns its result. apply's behavior is exactly the same as *func*(**args*,***keywords*). The * and ** forms are covered in Section 4.10 in Chapter 4. In almost all cases of practical interest, you can just use the syntax *func*(**args*,***keywords*) and avoid apply.

bool *Python 2.2 and later*

```
bool(x)
```

Returns 0, also known as False, if argument *x* evaluates as false; returns 1, also known as True, if argument *x* evaluates as true. See also Section 4.2.6 in Chapter 4. In Python 2.3, bool becomes a type (a subclass of int), and built-in names False and True refer to the only two instances of type bool. They are still numbers with values of 0 and 1 respectively, but str(True) becomes 'True', and str(False) becomes 'False', while in Python 2.2 the corresponding strings are '0' and '1' respectively.

8.3 The sys Module

The attributes of the sys module are bound to data and functions that provide information on the state of the Python interpreter or that affect the interpreter directly. This section documents the most frequently used attributes of sys, in alphabetical order.

argv

The list of command-line arguments passed to the main script. argv[0] is the name or full path of the main script, or '-c' if the -c option was used. See Section 8.4 later in this chapter for a good way to use sys.argv.

displayhook

```
displayhook(value)
```

In interactive sessions, the Python interpreter calls displayhook, passing it the result of each expression-statement entered. The default displayhook does nothing if *value* is None, otherwise it preserves and displays *value*:

```
if value is not None:
    _ _builtin_ _._ = value
    print repr(value)
```

You can rebind sys.displayhook in order to change interactive behavior. The original value is available as sys._ _displayhook_ _.

excepthook

```
excepthook(type,value,traceback
)
```

When an exception is not caught by any handler, Python calls excepthook, passing it the exception class, exception object, and traceback object, as covered in Chapter 6. The default excepthook displays the error and traceback. You can rebind sys.excepthook to change what is displayed for uncaught exceptions (just before Python returns to the interactive loop or terminates). The original value is also available as sys._ _excepthook_ _.

exc_info

```
exc_info(  )
```

If the current thread is handling an exception, exc_info returns a tuple whose three items are the class, object, and traceback for the exception. If the current thread is not handling any exception, exc_info returns (None,None,None). A traceback object indirectly holds references to all variables of all functions that propagated the exception. Thus, if you hold a reference to the traceback object (for example, indirectly, by binding a variable to the whole tuple that exc_info returns), Python has to retain in memory data that might otherwise be garbage-collected. So you should make sure that any binding to the traceback object is of short duration. To ensure that the binding gets removed, you can use a try/finally statement (discussed in Chapter 6).

exit

```
exit(arg=0)
```

8.4 The getopt Module

The getopt module helps parse the command-line options and arguments passed to a Python program, available in sys.argv. The getopt module distinguishes arguments proper from options: options start with '-' (or '--' for long-form options). The first non-option argument terminates option parsing (similar to most Unix commands, and differently from GNU and Windows commands). Module getopt supplies a single function, also called getopt.

getopt

```
getopt(args,options,
long_options=[  ])
```

Parses command-line options. *args* is usually sys.argv[1:]. *options* is a string: each character is an option letter, followed by ':' if the option takes a parameter. *long_options* is a list of strings, each a long-option name, without the leading '--', followed by '=' if the option takes a parameter.

When getopt encounters an error, it raises GetoptError, an exception class supplied by the getopt module. Otherwise, getopt returns a pair (*opts*,*args_proper*), where *opts* is a list of pairs of the form (*option*,*parameter*) in the same order in which options are found in *args*. Each *option* is a string that starts with a single hyphen for a short-form option or two hyphens for a long-form one; each *parameter* is also a string (an empty string for options that don't take parameters). *args_proper* is the list of program argument strings that are left after removing the options.

8.5 The copy Module

As discussed in Chapter 4, assignment in Python does not copy the right-hand side object being assigned. Rather, assignment adds a reference to the right-hand side object. When you want a copy of object *x*, you can ask *x* for a copy of itself. If *x* is a list, *x*[:] is a copy of *x*. If *x* is a dictionary, *x*.copy( ) returns a copy of *x*.

The copy module supplies a copy function that creates and returns a copy of most types of objects. Normal copies, such as *x*[:] for a list *x* and copy.copy(*x*), are also known as shallow copies. When *x* has references to other objects (e.g., items or attributes), a normal copy of *x* has distinct references to the same objects. Sometimes, however, you need a deep copy, where referenced objects are copied recursively. Module copy supplies a deepcopy(*x*) function that performs a deep copy and returns it as the function's result.

copy

```
copy(x)
```

Creates and returns a copy of *x* for *x* of most types (copies of modules, classes, frames, arrays, and internal types are not supported). If *x* is immutable, copy.copy(*x*) may return *x* itself as an optimization. A class can customize the way copy.copy copies its instances by having a special method __copy__(self) that returns a new object, a copy of self.

deepcopy

```
deepcopy(x,[memo])
```

Makes a deep copy of *x* and returns it. Deep copying implies a recursive walk over a directed graph of references. A precaution is needed to preserve the graph's shape: when references to the same object are met more than once during the walk, distinct copies must not be made. Rather, references to the same copied object must be used. Consider the following simple example:

```
sublist = [1,2]
original = [sublist, sublist]
thecopy = copy.deepcopy(original)
```

original[0] is original[1] is True (i.e., the two items of list original refer to the same object). This is an important property of original and therefore must be preserved in anything that claims to be a copy of it. The semantics of copy.deepcopy are defined to ensure that thecopy[0] is thecopy[1] is also True in this case. In other words, the shapes of the graphs of references of original and thecopy are the same. Avoiding repeated copying has an important beneficial side effect: preventing infinite loops that would otherwise occur if the graph has cycles.

copy.deepcopy accepts a second, optional argument *memo*, which is a dictionary that maps the id( ) of objects already copied to the new objects that are their copies. *memo* is passed by recursive calls of deepcopy to itself, but you may also explicitly pass it (normally as an originally empty dictionary) if you need to keep such a correspondence map between the identities of originals and copies of objects.

A class can customize the way copy.deepcopy copies its instances by having a special method __deepcopy__(self, *memo*) that returns a new object, a deep copy of self. When __deepcopy__ needs to deep copy some referenced object *subobject*, it must do so by calling copy.deepcopy(*subobject*,*memo*). When a class has no special method __deepcopy__, copy.deepcopy on an instance of that class tries to call special methods __getinitargs__, __getstate__, and __setstate__, which are covered in Section 11.1.2.3.

8.6 The bisect Module

The bisect module uses a bisection algorithm to keep a list in sorted order as items are inserted. bisect's operation is faster than calling a list's sort method after each insertion. This section documents the main functions supplied by bisect.

bisect

```
bisect(seq,item,lo=0,hi
=sys.maxint)
```

Returns the index *i* into *seq* where *item* should be inserted to keep *seq* sorted. In other words, *i* is such that each item in *seq*[:*i*] is less than or equal to *item*, and each item in *seq*[*i*:] is greater than or equal to *item*. *seq* must be a sorted sequence. For any sorted sequence *seq*, *seq*[bisect(*seq*,*y*)-1]= =*y* is equivalent to *y* in *seq*, but faster if len(*seq*) is large. You may pass optional arguments *lo* and *hi* to operate on the slice *seq*[*lo*:*hi*].

insort

```
insort(seq,item,lo=0,hi
=sys.maxint)
```

Like *seq*.insert(bisect(*seq*,*item*),*item*). In other words, *seq* must be a sorted mutable sequence, and insort modifies *seq* by inserting *item* at the right spot, so that *seq* remains sorted. You may pass optional arguments *lo* and *hi* to operate on the slice *seq*[*lo*:*hi*].

Module bisect also supplies functions bisect_left, bisect_right, insort_left, and insort_right for explicit control of search and insertion strategies into sequences that contain duplicates. bisect is a synonym for bisect_right, and insort is a synonym for insort_right.

8.7 The UserList, UserDict, and UserString Modules

The UserList, UserDict, and UserString modules each supply one class, with the same name as the respective module, that implements all the methods needed for the class's instances to be mutable sequences, mappings, and strings, respectively. When you need such polymorphism, you can subclass one of these classes and override some methods rather than have to implement everything yourself. In Python 2.2 and later, you can subclass built-in types list, dict, and str directly, to similar effect (see Section 5.2). However, these modules can still be handy if you need to create a classic class in order to keep your code compatible with Python 2.1 or earlier.

Each instance of one of these classes has an attribute called data that is a Python object of the corresponding built-in type (list, dict, and str, respectively). You can instantiate each class with an argument of the appropriate type (the argument is copied, so you can later modify it without side effects). UserList and UserDict can also be instantiated without arguments to create initially empty containers.

Module UserString also supplies class MutableString, which is very similar to class UserString except that instances of MutableString are mutable. Instances of MutableString and its subclasses cannot be keys into a dictionary. Instances of both UserString and MutableString can be Unicode strings rather than plain strings: just use a Unicode string as the initializer argument at instantiation time.

If you subclass UserList, UserDict, UserString, or MutableString and then override _ _init_ _, make sure the _ _init_ _ method you write can also be called with one argument of the appropriate type (as well as without arguments for UserList and UserDict). Also be sure that your _ _init_ _ method explicitly and appropriately calls the _ _init_ _ method of the superclass, as usual.

For maximum efficiency, you can arrange for your subclass to inherit from the appropriate built-in type when feasible (i.e., when your program runs with Python 2.2), but keep the ability to fall back to these modules when necessary (i.e., when your program runs with Python 2.1). Here is a typical idiom you can use for this purpose:

```
try:                            # can we subclass list?
    class _Temp(list):
        pass
except:                         # no: use UserList.UserList as base class
    from UserList import UserList as BaseList
else:                           # yes: remove _Temp and use list as base class
    del _Temp
    BaseList = list
class AutomaticallyExpandingList(BaseList):
    """a list such that you can always set L[i]=x even for a large i:
       L automatically grows, if needed, to make i a valid index."""
    def _ _setitem_ _(self, idx, val):
        self.extend((1+idx-len(self))*[None])
        BaseList._ _setitem_ _(self, idx, val)
```

Chapter 9. Strings and Regular Expressions

Python supports plain and Unicode strings extensively, with statements, operators, built-in functions, methods, and dedicated modules. This chapter covers the methods of string objects, talks about string formatting, documents the string, pprint, and repr modules, and discusses issues related to Unicode strings.

Regular expressions let you specify pattern strings and allow searches and substitutions. Regular expressions are not easy to master, but they are a powerful tool for processing text. Python offers rich regular expression functionality through the built-in re module, as documented in this chapter.

9.1 Methods of String Objects

Plain and Unicode strings are immutable sequences, as covered in Chapter 4. All immutable-sequence operations (repetition, concatenation, indexing, slicing) apply to strings. A string object *s* also supplies several non-mutating methods, as documented in this section. Unless otherwise noted, each method returns a plain string when *s* is a plain string, or a Unicode string when *s* is a Unicode string. Terms such as letters, whitespace, and so on refer to the corresponding attributes of the string module, covered later in this chapter. See also the later section Section 9.2.1.

capitalize

```
s.capitalize( )
```

Returns a copy of *s* where the first character, if a letter, is uppercase, and all other letters, if any, are lowercase.

center

```
s.center(n)
```

Returns a string of length max(len(*s*),*n*), with a copy of *s* in the central part, surrounded by equal numbers of spaces on both sides (e.g., 'ciao'.center(2) is 'ciao', 'ciao'.center(7) is ' ciao ').

count

```
s.count(sub,start=0,end
=sys.maxint)
```

Returns the number of occurrences of substring *sub* in *s*[*start*:*end*].

encode

```
s.encode(codec=None,errors
='strict')
```

Returns a plain string obtained from *s* with the given codec and error handling. See Section 9.6 later in this chapter for more details.

endswith

```
s.endswith(suffix,start=0,end
=sys.maxint)
```

Returns True when *s*[*start*:*end*] ends with *suffix*, otherwise False.

expandtabs

```
s.expandtabs(tabsize=8)
```

Returns a copy of *s* where each tab character is changed into one or more spaces, with tab stops every *tabsize* characters.

9.2 The string Module

The string module supplies functions that duplicate each method of string objects, as covered in the previous section. Each function takes the string object as its first argument. Module string also has several useful string-valued attributes:
ascii_letters

The string ascii_lowercase+ascii_uppercase
ascii_lowercase

The string 'abcdefghijklmnopqrstuvwxyz'
ascii_uppercase

The string 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
digits

The string '0123456789'
hexdigits

The string '0123456789abcdefABCDEF'
letters

The string lowercase+uppercase
lowercase

A string containing all characters that are deemed lowercase letters: at least 'abcdefghijklmnopqrstuvwxyz', but more letters (e.g., accented ones) may be present, depending on the active locale
octdigits

The string '01234567'
punctuation

The string '!"#$%&\'( )*+,-./:;<=>?@[\\]^_'{|}~' (i.e., all ASCII characters that are deemed punctuation characters in the "C" locale; does not depend on what locale is active)
printable

The string of those characters that are deemed printable (i.e., digits, letters, punctuation, and whitespace)
uppercase

A string containing all characters that are deemed uppercase letters: at least 'ABCDEFGHIJKLMNOPQRSTUVWXYZ', but more letters (e.g., accented ones) may be present, depending on the active locale
whitespace

A string containing all characters that are deemed whitespace: at least space, tab, linefeed, and carriage return, but more characters (e.g., control characters) may be present, depending on the active locale

You should not rebind these attributes, since other parts of the Python library may rely on them and the effects of rebinding them would be undefined.

9.2.1 Locale Sensitivity

The locale module is covered in Chapter 10. Locale setting affects some attributes of module string (letters,

9.3 String Formatting

In Python, a string-formatting expression has the syntax:

```
format % values
```

where *format* is a plain or Unicode string containing format specifiers and *values* is any single object or a collection of objects in a tuple or dictionary. Python's string-formatting operator has roughly the same set of features as the C language's printf and operates in a similar way. Each format specifier is a substring of *format* that starts with a percent sign (%) and ends with one of the conversion characters shown in Table 9-1.

Table 9-1. String-formatting conversion characters

Character	**Output format**	**Notes**
d, i	Signed decimal integer	Value must be number
u	Unsigned decimal integer	Value must be number
o	Unsigned octal integer	Value must be number
x	Unsigned hexadecimal integer (lowercase letters)	Value must be number
X	Unsigned hexadecimal integer (uppercase letters)	Value must be number
e	Floating-point value in exponential form (lowercase e for exponent)	Value must be number
E	Floating-point value in exponential form (uppercase E for exponent)	Value must be number
f, F	Floating-point value in decimal form	Value must be number
g, G	Like e or E when *exp* is greater than 4 or less than the precision; otherwise like f or F	*exp* is the exponent of the number being converted
c	Single character	Value can be integer or single-character string

9.4 The pprint Module

The pprint module pretty-prints complicated data structures, with formatting that may be more readable than that supplied by built-in function repr (see Chapter 8). To fine-tune the formatting, you can instantiate the PrettyPrinter class supplied by module pprint and apply detailed control, helped by auxiliary functions also supplied by module pprint. Most of the time, however, one of the two main functions exposed by module pprint suffices.

pformat

```
pformat(obj)
```

Returns a string representing the pretty-printing of *obj*.

pprint

```
pprint(obj,stream=sys.stdout)
```

Outputs the pretty-printing of *obj* to file object *stream*, with a terminating newline.

The following statements are the same:

```
 print pprint.pformat(x)
pprint.pprint(x)
```

Either of these constructs will be roughly the same as print *x* in many cases, such as when the string representation of *x* fits within one line. However, with something like *x*=range(30), print x displays x in two lines, breaking at an arbitrary point, while using module pprint displays x over 30 lines, one line per item. You can use module pprint when you prefer the module's specific display effects to the ones of normal string representation.

9.5 The repr Module

The repr module supplies an alternative to the built-in function repr (see Chapter 8), with limits on length for the representation string. To fine-tune the length limits, you can instantiate or subclass the Repr class supplied by module repr and apply detailed control. Most of the time, however, the main function exposed by module repr suffices.

repr

```
repr(obj)
```

Returns a string representing *obj*, with sensible limits on length.

9.6 Unicode

Plain strings are converted into Unicode strings either explicitly, with the unicode built-in, or implicitly, when you pass a plain string to a function that expects Unicode. In either case, the conversion is done by an auxiliary object known as a *codec* (for coder-decoder). A codec can also convert Unicode strings to plain strings either explicitly, with the encode method of Unicode strings, or implicitly.

You identify a codec by passing the codec name to unicode or encode. When you pass no codec name and for implicit conversion, Python uses a default encoding, normally 'ascii'. (You can change the default encoding in the startup phase of a Python program, as covered in Chapter 13; see also setdefaultencoding in Chapter 8.) Every conversion has an explicit or implicit argument *errors*, a string specifying how conversion errors are to be handled. The default is 'strict', meaning any error raises an exception. When *errors* is 'replace', the conversion replaces each character causing an error with '?' in a plain-string result or with u'\ufffd' in a Unicode result. When *errors* is 'ignore', the conversion silently skips characters that cause errors.

9.6.1 The codecs Module

The mapping of codec names to codec objects is handled by the codecs module. This module lets you develop your own codec objects and register them so that they can be looked up by name, just like built-in codecs. Module codecs also lets you look up any codec explicitly, obtaining the functions the codec uses for encoding and decoding, as well as factory functions to wrap file-like objects. Such advanced facilities of module codecs are rarely used, and are not covered further in this book.

The codecs module, together with the encodings package, supplies built-in codecs useful to Python developers dealing with internationalization issues. Any supplied codec can be installed as the default by module sitecustomize, or can be specified by name when converting explicitly between plain and Unicode strings. The codec normally installed by default is 'ascii', which accepts only characters with codes between 0 and 127, the 7-bit range of the American Standard Code for Information Interchange (ASCII) that is common to most encodings. A popular codec is 'latin-1', a fast, built-in implementation of the ISO 8859-1 encoding that offers a one-byte-per-character encoding of all special characters needed for Western European languages.

The codecs module also supplies codecs implemented in Python for most ISO 8859 encodings, with codec names from 'iso8859-1' to 'iso8859-15'. On Windows systems only, the codec named 'mbcs' wraps the platform's multibyte character set conversion procedures. In Python 2.2, many codecs are added to support Asian languages. Module codecs also supplies several standard code pages (codec names from 'cp037' to 'cp1258'), Mac-specific encodings (codec names from 'mac-cyrillic' to 'mac-turkish'), and Unicode standard encodings 'utf-8' and 'utf-16' (the latter also have specific big-endian and little-endian variants 'utf-16-be' and 'utf-16-le'). For use with UTF-16, module codecs also supplies attributes BOM_BE and BOM_LE, byte-order marks for big-endian and little-endian machines respectively, and BOM, byte-order mark for the current platform.

Module codecs also supplies two functions to make it easier to deal with encoded text during input/output operations.

EncodedFile

```
EncodedFile(file,datacodec,
filecodec=None,errors='strict')
```

Wraps the file-like object *file*, returning another file-like object *ef* that implicitly and transparently applies the given encodings to all data read from or written to the file. When you write a string *s* to *ef*, *ef* first decodes *s* with the codec

9.7 Regular Expressions and the re Module

A regular expression is a string that represents a pattern. With regular expression functionality, you can compare that pattern to another string and see if any part of the string matches the pattern.

The re module supplies all of Python's regular expression functionality. The compile function builds a regular expression object from a pattern string and optional flags. The methods of a regular expression object look for matches of the regular expression in a string and/or perform substitutions. Module re also exposes functions equivalent to a regular expression's methods, but with the regular expression's pattern string as their first argument.

Regular expressions can be difficult to master, and this book does not purport to teach them—I cover only the ways in which you can use them in Python. For general coverage of regular expressions, I recommend the book Mastering Regular Expressions, by Jeffrey Friedl (O'Reilly). Friedl's book offers thorough coverage of regular expressions at both the tutorial and advanced levels.

9.7.1 Pattern-String Syntax

The pattern string representing a regular expression follows a specific syntax:

- Alphabetic and numeric characters stand for themselves. A regular expression whose pattern is a string of letters and digits matches the same string.
- Many alphanumeric characters acquire special meaning in a pattern when they are preceded by a backslash (\).
- Punctuation works the other way around. A punctuation character is self-matching when escaped, and has a special meaning when unescaped.
- The backslash character itself is matched by a repeated backslash (i.e., the pattern \\).

Since regular expression patterns often contain backslashes, you generally want to specify them using raw-string syntax (covered in Chapter 4). Pattern elements (e.g., r'\t', which is equivalent to the non-raw string literal '\\t') do match the corresponding special characters (e.g., the tab character '\t'). Therefore, you can use raw-string syntax even when you do need a literal match for some such special character.

Table 9-2 lists the special elements in regular expression pattern syntax. The exact meanings of some pattern elements change when you use optional flags, together with the pattern string, to build the regular expression object. The optional flags are covered later in this chapter.

Table 9-2. Regular expression pattern syntax

Part III: Python Library and Extension Modules

Chapter 10. File and Text Operations

This chapter covers dealing with files and the filesystem in Python. A file is a stream of bytes that a program can read and/or write, while a filesystem is a hierarchical repository of files on a particular computer system. Because files are such a core programming concept, several other chapters also contain material about handling files of specific kinds.

In Python, the os module supplies many of the functions that operate on the filesystem, so this chapter starts by introducing the os module. The chapter then proceeds to cover operations on the filesystem, including comparing, copying, and deleting directories and files, working with file paths, and accessing low-level file descriptors.

Next, this chapter discusses the typical ways Python programs read and write data, via built-in file objects and the polymorphic concept of file-like objects (i.e., objects that are not files, but still behave to some extent like files). Python file objects directly support the concept of text files, which are streams of characters encoded as bytes. The chapter also covers Python's support for data in compressed form, such as archives in the popular ZIP format.

While many modern programs rely on a graphical user interface (GUI), text-based, non-graphical user interfaces are often still useful, as they are simple, fast to program, and lightweight. This chapter concludes with material about text input and output in Python, including information about presenting text that is understandable to different users, no matter where they are or what language they speak. This is known as internationalization (often abbreviated i18n).

10.1 The os Module

The os module is an umbrella module that presents a reasonably uniform cross-platform view of the different capabilities of various operating systems. The module provides functionality for creating files, manipulating files and directories, and creating, managing, and destroying processes. This chapter covers the filesystem-related capabilities of the os module, while Chapter 14 covers the process-related capabilities.

The os module supplies a name attribute, which is a string that identifies the kind of platform on which Python is being run. Possible values for name are 'posix' (all kinds of Unix-like platforms), 'nt' (all kinds of 32-bit Windows platforms), 'mac', 'os2', and 'java'. You can often exploit unique capabilities of a platform, at least in part, through functions supplied by os. This book deals with cross-platform programming, however, not with platform-specific functionality, so I do not cover parts of os that exist only on one kind of platform, nor do I cover platform-specific modules. All functionality covered in this book is available at least on both 'posix' and 'nt' platforms. However, I do cover any differences among the ways in which each given piece of functionality is provided on different platforms.

10.1.1 OSError Exceptions

When a request to the operating system fails, os raises an exception, an instance of OSError. os also exposes class OSError with the name os.error. Instances of OSError expose three useful attributes:
errno

The numeric error code of the operating system error
strerror

A string that summarily describes the error
filename

The name of the file on which the operation failed (for file-related functions only)

os functions can also raise other standard exceptions, typically TypeError or ValueError, when the error is that they have been called with invalid argument types or values and the underlying operating system functionality has not even been attempted.

10.1.2 The errno Module

The errno module supplies symbolic names for error code numbers. To handle possible system errors selectively, based on error codes, use errno to enhance your program's portability and readability. For example, here's how you might handle only "file not found" errors, while propagating others:

```
try: os.some_os_function_or_other( )
except OSError, err:
    import errno
    # check for "file not found" errors
    if err.errno != errno.ENOENT: raise               # reraise other cases
    # proceed with the specific case you can handle
    print "Warning: file", err.filename, "not found -- continuing"
```

errno also supplies a dictionary named errorcode: the keys are error code numbers, and the corresponding names are the error names, such as 'ENOENT'. Displaying errno.errorcode[err.errno], as part of your diagnosis of some os.error instance err, can often make diagnosis clearer and more understandable to readers who are specialists of the specific platform.

10.2 Filesystem Operations

Using the os module, you can manipulate the filesystem in a variety of ways: creating, copying, and deleting files and directories, comparing files, and examining filesystem information about files and directories. This section documents the attributes and methods of the os module that you use for these purposes, and also covers some related modules that operate on the filesystem.

10.2.1 Path-String Attributes of the os Module

A file or directory is identified by a string, known as its *path*, whose syntax depends on the platform. On both Unix-like and Windows platforms, Python accepts Unix syntax for paths, with slash (/) as the directory separator. On non-Unix-like platforms, Python also accepts platform-specific path syntax. On Windows, for example, you can use backslash (\) as the separator. However, you do need to double up each backslash to \\ in normal string literals or use raw-string syntax as covered in Chapter 4. In the rest of this chapter, for brevity, Unix syntax is assumed in both explanations and examples.

Module os supplies attributes that provide details about path strings on the current platform. You should typically use the higher-level path manipulation operations covered in Section 10.2.4 later in this chapter, rather than lower-level string operations based on these attributes. However, the attributes may still be useful at times:
curdir

The string that denotes the current directory ('.' on Unix and Windows)
defpath

The default search path used if the environment lacks a PATH environment variable
linesep

The string that terminates text lines ('\n' on Unix, '\r\n' on Windows)
extsep

The string that separates the extension part of a file's name from the rest of the name ('.' on Unix and Windows)
pardir

The string that denotes the parent directory ('..' on Unix and Windows)
pathsep

The separator between paths in lists of paths, such as those used for the environment variable PATH (':' on Unix, ';' on Windows)
sep

The separator of path components ('/' on Unix, '\\' on Windows)

10.2.2 Permissions

Unix-like platforms associate nine bits with each file or directory, three each for the file's owner (user), its group, and anybody else, indicating whether the file or directory can be read, written, and executed by the specified subject. These nine bits are known as the file's *permission bits*, part of the file's *mode* (a bit string that also includes other bits describing the file). These bits are often displayed in octal notation, which groups three bits in each digit. For example, a mode of 0664 indicates a file that can be read and written by its owner and group, but only read, not written, by anybody else. When any process on a Unix-like system creates a file or directory, the operating system applies to the specified mode a bit mask known as the process's *umask*, which can remove some of the permission bits.

10.3 File Objects

As discussed earlier in this chapter, file is a built-in type in Python. With a file object, you can read and/or write data to a file as seen by the underlying operating system. Python reacts to any I/O error related to a file object by raising an instance of built-in exception class IOError. Errors that cause this exception include open failing to open or create a file, calling a method on a file object to which that method doesn't apply (e.g., calling write on a read-only file object or calling seek on a non-seekable file), and I/O errors diagnosed by a file object's methods. This section documents file objects, as well as some auxiliary modules that help you access and deal with their contents.

10.3.1 Creating a File Object with open

You normally create a Python file object with the built-in open, which has the following syntax:

```
open(filename,mode='r',bufsize=-1)
```

open opens the file named by *filename*, which must be a string that denotes any path to a file. open returns a Python file object, which is an instance of the built-in type file. Calling file is just like calling open, but file was first introduced in Python 2.2. If you explicitly pass a *mode* string, open can also create *filename* if the file does not already exist (depending on the value of *mode*, as we'll discuss in a moment). In other words, despite its name, open is not limited to opening existing files, but is also able to create new ones if needed.

10.3.1.1 File mode

mode is a string that denotes how the file is to be opened (or created). mode can have the following values:

'r'

The file must already exist, and it is opened in read-only mode.

'w'

The file is opened in write-only mode. The file is truncated and overwritten if it already exists, or created if it does not exist.

'a'

The file is opened in write-only mode. The file is kept intact if it already exists, and the data you write is appended to what's already in the file. The file is created if it does not exist. Calling *f*.seek is innocuous, but has no effect.

'r+'

The file must already exist and is opened for both reading and writing, so all methods of *f* can be called.

'w+'

The file is opened for both reading and writing, so all methods of *f* can be called. The file is truncated and overwritten if it already exists, or created if it does not exist.

'a+'

The file is opened for both reading and writing, so all methods of *f* can be called. The file is kept intact if it already exists, and the data you write is appended to what's already in the file. The file is created if it does not exist. Calling *f* .seek has no effect if the next I/O operation on *f* writes data, but works normally if the next I/O operation on *f* reads data.

10.3.1.2 Binary and text modes

The *mode* string may also have any of the values just explained followed by a b or t. b denotes binary mode, while t

10.4 Auxiliary Modules for File I/O

File objects supply all functionality that is strictly needed for file I/O. There are some auxiliary Python library modules, however, that offer convenient supplementary functionality, making I/O even easier and handier in several important special cases.

10.4.1 The fileinput Module

The fileinput module lets you loop over all the lines in a list of text files. Performance is quite good, comparable to the performance of direct iteration on each file, since fileinput uses internal buffering to minimize I/O. Therefore, you can use module fileinput for line-oriented file input whenever you find the module's rich functionality convenient, without worrying about performance. The input function is the main function of module fileinput, and the module also provides a FileInput class that supports the same functionality as the module's functions.

close

```
close(  )
```

Closes the whole sequence, so that iteration stops and no file remains open.

FileInput

```
class FileInput(files=None,
inplace=0,backup='',bufsize=0)
```

Creates and returns an instance *f* of class FileInput. Arguments are the same as for fileinput.input, and methods of *f* have the same names, arguments, and semantics as functions of module fileinput. *f* also supplies a method readline, which reads and returns the next line. You can use class FileInput explicitly, rather than the single implicit instance used by the functions of module fileinput, when you want to nest or otherwise mix loops that read lines from more than one sequence of files.

filelineno

```
filelineno(  )
```

Returns the number of lines read so far from the file now being read. For example, returns 1 if the first line has just been read from the current file.

filename

```
filename(  )
```

Returns the name of the file being read, or None if no line has been read yet.

input

```
input(files=None,inplace=0,
backup='',bufsize=0)
```

10.5 The StringIO and cStringIO Modules

You can implement file-like objects by writing Python classes that supply the methods you need. If all you want is for data to reside in memory rather than on a file as seen by the operating system, you can use the StringIO or cStringIO module. The two modules are almost identical: each supplies a factory function to create in-memory file-like objects. The difference between them is that objects created by module StringIO are instances of class StringIO.StringIO. You may inherit from this class to create your own customized file-like objects, overriding the methods that you need to specialize. Objects created by module cStringIO, on the other hand, are instances of a special-purpose type, not of a class. Performance is much better when you can use cStringIO, but inheritance is not feasible. Furthermore, cStringIO does not support Unicode.

Each module supplies a factory function named StringIO that creates a file-like object *fl*.

StringIO

```
StringIO(str='')
```

Creates and returns an in-memory file-like object *fl*, with all methods and attributes of a built-in file object. The data contents of *fl* are initialized to be a copy of argument *str*, which must be a plain string for the StringIO factory function in cStringIO, while it can be a plain or Unicode string for the function in StringIO.

Besides all methods and attributes of built-in file objects, as covered in Section 10.3.2 earlier in this chapter, *fl* supplies one supplementary method, getvalue.

getvalue

```
fl.

getvalue(  )
```

Returns the current data contents of *fl* as a string. You cannot call *fl*.getvalue after you call *fl*.close: close frees the buffer that *fl* internally keeps, and getvalue needs to access the buffer to yield its result.

10.6 Compressed Files

Although storage space and transmission bandwidth are increasingly cheap and abundant, in many cases you can save such resources, at the expense of some computational effort, by using compression. Since computational power grows cheaper and more abundant even faster than other resources, such as bandwidth, compression's popularity keeps growing. Python makes it easy for your programs to support compression by supplying dedicated modules for compression as part of every Python distribution.

10.6.1 The gzip Module

The gzip module lets you read and write files compatible with those handled by the powerful GNU compression programs *gzip* and *gunzip*. The GNU programs support several compression formats, but module gzip supports only the highly effective native *gzip* format, normally denoted by appending the extension *.gz* to a filename. Module gzip supplies the GzipFile class and an open factory function.

GzipFile

```
class GzipFile(filename=None,
mode=None,compresslevel=9,
               fileobj=None)
```

Creates and returns a file-like object *f* that wraps the file or file-like object *fileobj*. *f* supplies all methods of built-in file objects except seek and tell. Thus, *f* is not seekable: you can only access *f* sequentially, whether for reading or writing. When *fileobj* is None, *filename* must be a string that names a file: GzipFile opens that file with the given *mode* (by default, 'rb'), and *f* wraps the resulting file object. *mode* should be one of 'ab', 'rb', 'wb', or None. If *mode* is None, *f* uses the mode of *fileobj* if it is able to find out the mode; otherwise it uses 'rb'. If *filename* is None, *f* uses the filename of *fileobj* if able to find out the name; otherwise it uses ''. *compresslevel* is an integer between 1 and 9: 1 requests modest compression but fast operation, and 9 requests the best compression feasible, even if that requires more computation.

File-like object *f* generally delegates all methods to the underlying file-like object *fileobj*, transparently accounting for compression as needed. However, *f* does not allow non-sequential access, so *f* does not supply methods seek and tell. Moreover, calling *f*.close does not close *fileobj* when *f* was created with an argument *fileobj* that is not None. This behavior of *f*.close is very important when *fileobj* is an instance of StringIO.StringIO, since it means you can call *fileobj*.getvalue after *f*.close to get the compressed data as a string. This behavior also means that you have to call *fileobj*.close explicitly after calling *f*.close.

open

```
open(filename,mode='rb',
compresslevel=9)
```

Like GzipFile(*filename*,*mode*,*compresslevel*), but *filename* is mandatory and there is no provision for passing an already opened *fileobj*.

Say that you have some function *f*(*x*) that writes data to a text file object *x*, typically by calling *x*.write and/or *x*.writelines. Getting *f* to write data to a *gzip*-compressed text file instead is easy:

```
 import gzip
underlying_file = open('x.txt.gz', 'wb')
compressing_wrapper = gzip.GzipFile(fileobj=underlying_file, mode='wt')
```

10.7 Text Input and Output

Python presents non-GUI text input and output channels to your programs as file objects, so you can use the methods of file objects (covered in Section 10.3 earlier in this chapter) to manipulate these channels.

10.7.1 Standard Output and Standard Error

The sys module, covered in Chapter 8, has attributes stdout and stderr, file objects to which you can write. Unless you are using some sort of shell redirection, these streams connect to the terminal in which your script is running. Nowadays, actual terminals are rare: the terminal is generally a screen window that supports text input/output (e.g., an MS-DOS Prompt console on Windows or an *xterm* window on Unix).

The distinction between sys.stdout and sys.stderr is a matter of convention. sys.stdout, known as your script's standard output, is where your program emits results. sys.stderr, known as your script's standard error, is where error messages go. Separating program results from error messages helps you use shell redirection effectively. Python respects this convention, using sys.stderr for error and warning messages.

10.7.2 The print Statement

Programs that output results to standard output often need to write to sys.stdout. Python's print statement can be a convenient alternative to sys.stdout.write. The print statement has the following syntax:

```
print [>>fileobject,] expressions [,]
```

The normal destination of print's output is the file or file-like object that is the value of the stdout attribute of the sys module. However, when >>*fileobject*, is present right after keyword print, the statement uses the given *fileobject* instead of sys.stdout. *expressions* is a list of zero or more expressions separated by commas (,). print outputs each expression, in order, as a string (using the built-in str, covered in Chapter 8), with a space to separate strings. After all expressions, print by default outputs '\n' to terminate the line. When a trailing comma is present at the end of the statement, however, print does not output the closing '\n'.

print works well for the kind of informal output used during development to help you debug your code. For production output, you often need more control of formatting than print affords. You may need to control spacing, field widths, the number of decimals for floating-point values, and so on. In this case, prepare the output as a string with the string-formatting operator % covered in Chapter 9. Then, you can output the resulting string, normally with the write method of the appropriate file object.

When you want to direct print's output to another file, you can temporarily change sys.stdout. The following example shows a general-purpose redirection function that you can use for such a temporary change:

```
def redirect(func, *args, **kwds):
    """redirect(func, ...) -> (output string result, func's return value)

    func must be a callable that outputs results to standard output.
    redirect captures those results in memory and returns a pair, with
    the results as the first item and func's return value as the second
    one.
    """
    import sys, cStringIO
    save_out = sys.stdout
    sys.stdout = cStringIO.StringIO(  )
    try:
        retval = func(*args, **kwds)
```

10.8 Richer-Text I/O

The tools we have covered so far support the minimal subset of text I/O functionality that all platforms supply. Most platforms also offer richer-text I/O capabilities, such as responding to single keypresses (not just to entire lines of text) and showing text in any spot of the terminal (not just sequentially).

Python extensions and core Python modules let you access platform-specific functionality. Unfortunately, various platforms expose this functionality in different ways. To develop cross-platform Python programs with rich-text I/O functionality, you may need to wrap different modules uniformly, importing platform-specific modules conditionally (usually with the try/except idiom covered in Chapter 6).

10.8.1 The readline Module

The readline module wraps the GNU Readline Library. Readline lets the user edit text lines during interactive input, and also recall previous lines for further editing and re-entry. GNU Readline is widely installed on Unix-like platforms, and is available at http://cnswww.cns.cwru.edu/~chet/readline/rltop.html. A Windows port (http://starship.python.net/crew/kernr/) is available, but not widely deployed. Chris Gonnerman's module, Alternative Readline for Windows, implements a subset of Python's standard readline module (using a small dedicated *.pyd* file instead of GNU Readline) and can be freely downloaded from http://newcenturycomputers.net/projects/readline.html .

When either readline module is loaded, Python uses Readline for all line-oriented input, such as raw_input. The interactive Python interpreter always tries loading readline to enable line editing and recall for interactive sessions. You can call functions supplied by module readline to control advanced functionality, particularly the history functionality for recalling lines entered in previous sessions, and the completion functionality for context-sensitive completion of the word being entered. See http://cnswww.cns.cwru.edu/~chet/readline/rltop.html#Documentation for GNU Readline documentation, with details on configuration commands. Alternative Readline also supports history, but the completion-related functions it supplies are dummy ones: these functions don't perform any operation, and exist only for compatibility with GNU Readline.

get_history_length

```
get_history_length(  )
```

Returns the number of lines of history that are saved to the history file. When the returned value is less than 0, all lines in the history are saved.

parse_and_bind

```
parse_and_bind(readline_cmd)
```

Gives Readline a configuration command. To let the user hit Tab to request completion, call parse_and_bind('tab: complete'). See the GNU Readline documentation for other useful values of *readline_cmd*.

read_history_file

```
read_history_file(filename
='~/.history')
```

10.9 Interactive Command Sessions

The cmd module offers a simple way to handle interactive sessions of commands. Each command is a line of text. The first word of each command is a verb defining the requested action. The rest of the line is passed as an argument to the method that implements the action that the verb requests.

Module cmd supplies class Cmd to use as a base class, and you define your own subclass of cmd.Cmd. The subclass supplies methods with names starting with do_ and help_, and may also optionally override some of Cmd's methods. When the user enters a command line such as *verb and the rest*, as long as the subclass defines a method named do_*verb*, Cmd.onecmd calls:

```
self.do_verb('and the rest')
```

Similarly, as long as the subclass defines a method named help_*verb*, Cmd.do_help calls it when the command line starts with either 'help *verb*' or '?*verb*'. Cmd, by default, also shows suitable error messages if the user tries to use, or asks for help about, a verb for which the subclass does not define a needed method.

10.9.1 Methods of Cmd Instances

An instance *c* of a subclass of class Cmd supplies the following methods (many of these methods are meant to be overridden by the subclass).

cmdloop

```
c.cmdloop(intro=None)
```

Performs an entire interactive session of line-oriented commands. cmdloop starts by calling *c*.preloop( ), then outputs string *intro* (*c*.intro, if *intro* is None). Then *c*.cmdloop enters a loop. In each iteration of the loop, cmdloop reads line *s* with *s*=raw_input(*c*.prompt). When standard input reaches end-of-file, cmdloop sets *s*='EOF'. If *s* is not 'EOF', cmdloop preprocesses string *s* with *s*=*c*.precmd(*s*), then calls *flag*=*c*.onecmd(*s*). When onecmd returns a true value, this is a tentative request to terminate the command loop. Now cmdloop calls *flag*=*c*.postcmd(*flag*,*s*) to check if the loop should terminate. If *flag* is now true, the loop terminates; otherwise another iteration of the loop executes. If the loop is to terminate, cmdloop calls *c*.postloop( ), then terminates. This structure of cmdloop is probably easiest to understand by showing Python code equivalent to the method just described:

```
def cmdloop(self, intro=None):
    self.preloop(  )
    if intro is None: intro = self.intro
    print intro
    while True:
        try: s = raw_input(self.prompt)
        except EOFError: s = `EOF'
        else: s = self.precmd(s)
        flag = self.onecmd(s)
        flag = self.postcmd(flag, s)
        if flag: break
    self.postloop(  )
```

cmdloop is a good example of the design pattern known as Template Method. Such a method performs little substantial work itself; rather, it structures and organizes calls to other methods. Subclasses may override the other methods, to define the details of class behavior within the overall framework thus established. When you inherit from Cmd, you almost never override method cmdloop, since cmdloop's structure is the main thing you get by subclassing Cmd.

default

10.10 Internationalization

Most programs present some information to users as text. Such text should be understandable and acceptable to the user. For example, in some countries and cultures, the date "March 7" can be concisely expressed as "3/7". Elsewhere, "3/7" indicates "July 3", and the string that means "March 7" is "7/3". In Python, such cultural conventions are handled with the help of standard module locale.

Similarly, a greeting can be expressed in one natural language by the string "Benvenuti", while in another language the string to use is "Welcome". In Python, such translations are handled with the help of standard module gettext.

Both kinds of issues are commonly called *internationalization* (often abbreviated *i18n*, as there are 18 letters between i and n in the full spelling). This is actually a misnomer, as the issues also apply to programs used within one nation by users of different languages or cultures.

10.10.1 The locale Module

Python's support for cultural conventions is patterned on that of C, slightly simplified. In this architecture, a program operates in an environment of cultural conventions known as a *locale*. The locale setting permeates the program and is typically set early on in the program's operation. The locale is not thread-specific, and module locale is not thread-safe. In a multithreaded program, set the program's locale before starting secondary threads.

If a program does not call locale.setlocale, the program operates in a neutral locale known as the C locale. The C locale is named from this architecture's origins in the C language, and is similar, but not identical, to the U.S. English locale. Alternatively, a program can find out and accept the user's default locale. In this case, module locale interacts with the operating system (via the environment, or in other system-dependent ways) to establish the user's preferred locale. Finally, a program can set a specific locale, presumably determining which locale to set on the basis of user interaction, or via persistent configuration settings such as a program initialization file.

A locale setting is normally performed across the board, for all relevant categories of cultural conventions. This wide-spectrum setting is denoted by the constant attribute LC_ALL of module locale. However, the cultural conventions handled by module locale are grouped into categories, and in some cases a program can choose to mix and match categories to build up a synthetic composite locale. The categories are identified by the following constant attributes of module locale:
LC_COLLATE

String sorting: affects functions strcoll and strxfrm in locale
LC_CTYPE

Character types: affects aspects of module string (and string methods) that have to do with letters, lowercase, and uppercase
LC_MESSAGES

Messages: may affect messages displayed by the operating system, for example function os.strerror and module gettext
LC_MONETARY

Formatting of currency values: affects function locale.localeconv
LC_NUMERIC

Chapter 11. Persistence and Databases

Python supports a variety of ways of making data persistent. One such way, known as serialization, involves viewing the data as a collection of Python objects. These objects can be saved, or serialized, to a byte stream, and later loaded and recreated, or deserialized, back from the byte stream. Object persistence layers on top of serialization and adds such features as object naming. This chapter covers the built-in Python modules that support serialization and object persistence.

Another way to make data persistent is to store it in a database. One simple type of database is actually just a file format that uses keyed access to enable selective reading and updating of relevant parts of the data. Python supplies modules that support several variations of this file format, known as DBM, and these modules are covered in this chapter.

A relational database management system (RDBMS), such as MySQL or Oracle, provides a more powerful approach to storing, searching, and retrieving persistent data. Relational databases rely on dialects of Structured Query Language (SQL) to create and alter a database's schema, insert and update data in the database, and query the database according to search criteria. This chapter does not provide any reference material on SQL. For that purpose, I recommend SQL in a Nutshell, by Kevin Kline (O'Reilly). Unfortunately, despite the existence of SQL standards, no two RDBMSes implement exactly the same SQL dialect.

The Python standard library does not come with an RDBMS interface. However, many free third-party modules let your Python programs access a specific RDBMS. Such modules mostly follow the Python Database API 2.0 standard, also known as the DBAPI. This chapter covers the DBAPI standard and mentions some of the third-party modules that implement it.

11.1 Serialization

Python supplies a number of modules that deal with I/O operations that serialize (save) entire Python objects to various kinds of byte streams, and deserialize (load and recreate) Python objects back from such streams. Serialization is also called *marshaling*.

11.1.1 The marshal Module

The marshal module supports the specific serialization tasks needed to save and reload compiled Python files (*.pyc* and *.pyo*). marshal only handles instances of fundamental built-in data types: None, numbers (plain and long integers, float, complex), strings (plain and Unicode), code objects, and built-in containers (tuples, lists, dictionaries) whose items are instances of elementary types. marshal does not handle instances of user-defined types, nor classes and instances of classes. marshal is faster than other serialization modules. Code objects are supported only by marshal, not by other serialization modules. Module marshal supplies the following functions.

dump, dumps

```
dump(value,fileobj)
dumps(value)
```

dumps returns a string representing object *value*. dump writes the same string to file object *fileobj*, which must be opened for writing in binary mode. dump(*v*,*f*) is just like *f*.write(dumps(*v*)). *fileobj* cannot be a file-like object: it must be an instance of type file.

load, loads

```
load(fileobj)
loads(str)
```

loads creates and returns the object *v* previously dumped to string *str*, so that, for any object *v* of a supported type, *v* equals loads(dumps(*v*)). If *str* is longer than dumps(*v*), loads ignores the extra bytes. load reads the right number of bytes from file object *fileobj*, which must be opened for reading in binary mode, and creates and returns the object *v* represented by those bytes. *fileobj* cannot be a file-like object: it must be an instance of type file.

Functions load and dump are complementary. In other words, a sequence of calls to load(*f*) deserializes the same values previously serialized when *f*'s contents were created by a sequence of calls to dump(*v*,*f*). Objects that are dumped and loaded in this way can be instances of any mix of supported types.

Suppose you need to analyze several text files, whose names are given as your program's arguments, and record where each word appears in those files. The data you need to record for each word is a list of (*filename*, *line-number*) pairs. The following example uses marshal to encode lists of (*filename*, *line-number*) pairs as strings and store them in a DBM-like file (as covered later in this chapter). Since those lists contain tuples, each made up of a string and a number, they are within marshal's abilities to serialize.

```
import fileinput, marshal, anydbm
wordPos = {  }
for line in fileinput.input(  ):
    pos = fileinput.filename(  ), fileinput.filelineno(  )
    for word in line.split(  ):
        wordPos.setdefault(word,[  ]).append(pos)
dbmOut = anydbm.open('indexfilem','n')
```

11.2 DBM Modules

A DBM-like file is a file that contains a set of pairs of strings (*key*,*data*), with support for fetching or storing the data given a key, known as *keyed access*. DBM-like files were originally supported on early Unix systems, with functionality roughly equivalent to that of access methods popular on other mainframe and minicomputers of the time, such as ISAM, the Indexed-Sequential Access Method. Today, several different libraries, available for many platforms, let programs written in many different languages create, update, and read DBM-like files.

Keyed access, while not as powerful as the data access functionality of relational databases, may often suffice for a program's needs. And if DBM-like files are sufficient, you may end up with a program that is smaller, faster, and more portable than one that uses an RDBMS.

The classic *dbm* library, whose first version introduced DBM-like files many years ago, has limited functionality, but tends to be available on most Unix platforms. The GNU version, *gdbm*, is richer and also widespread. The BSD version, *dbhash*, offers superior functionality. Python supplies modules that interface with each of these libraries if the relevant underlying library is installed on your system. Python also offers a minimal DBM module, dumbdbm (usable anywhere, as it does not rely on other installed libraries), and generic DBM modules, which are able to automatically identify, select, and wrap the appropriate DBM library to deal with an existing or new DBM file. Depending on your platform, your Python distribution, and what *dbm*-like libraries you have installed on your computer, the default Python build may install some subset of these modules. In general, at a minimum, you can rely on having module dbm on Unix-like platforms, module dbhash on Windows, and dumbdbm on any platform.

11.2.1 The anydbm Module

The anydbm module is a generic interface to any other DBM module. anydbm supplies a single factory function.

open

```
open(filename,flag='r',mode
=0666)
```

Opens or creates the DBM file named by *filename* (a string that can denote any path to a file, not just a name), and returns a suitable mapping object corresponding to the DBM file. When the DBM file already exists, open uses module whichdb to determine which DBM library can handle the file. When open creates a new DBM file, open chooses the first available DBM module in order of preference: dbhash, gdbm, dbm, and dumbdbm.

flag is a one-character string that tells open how to open the file and whether to create it, as shown in Table 11-1. *mode* is an integer that open uses as the file's permission bits if open creates the file, as covered in Section 10.2.2 in Chapter 10. Not all DBM modules use *flags* and *mode*, but for portability's sake you should always supply appropriate values for these arguments when you call anydbm.open.

Table 11-1. flag values for anydbm.open

Flag	**Read-only?**	**If file exists**	**If file does not exist**
`'r'`	yes	open opens the file	open raises error
`'w'`	no	open opens the file	open raises error

11.3 The Berkeley DB Module

Python comes with the bsddb module, which wraps the Berkeley Database library (also known as BSD DB) if that library is installed on your system and your Python installation is built to support it. With the BSD DB library, you can create hash, binary tree, or record-based files that generally behave like dictionaries. On Windows, Python includes a port of the BSD DB library, thus ensuring that module bsddb is always usable. To download BSD DB sources, binaries for other platforms, and detailed documentation on BSD DB, see http://www.sleepycat.com. Module bsddb supplies three factory functions, btopen, hashopen, and rnopen.

btopen, hashopen, rnopen

```
btopen(filename,flag='r',*
many_other_optional_arguments)
hashopen(filename,flag='r',*
many_other_optional_arguments)
rnopen(filename,flag='r',*
many_other_optional_arguments)
```

btopen opens or creates the binary tree format file named by *filename* (a string that denotes any path to a file, not just a name), and returns a suitable BTree object to access and manipulate the file. Argument *flag* has exactly the same values and meaning as for anydbm.open. Other arguments indicate low-level options that allow fine-grained control, but are rarely used.

hashopen and rnopen work the same way, but open or create hash format and record format files, returning objects of type Hash and Record. hashopen is generally the fastest format and makes sense when you are using keys to look up records. However, if you also need to access records in sorted order, use btopen, or if you need to access records in the same order in which you originally wrote them, use rnopen. Using hashopen does not keep records in order in the file.

An object *b* of any of the types BTree, Hash, and Record can be indexed as a mapping, with both keys and values constrained to being strings. Further, *b* also supports sequential access through the concept of a current record. *b* supplies the following methods.

close

```
b.close(  )
```

Closes *b*. Call no other method on *b* after *b*.close( ).

first

```
b.first(  )
```

Sets *b*'s current record to the first record, and returns a pair (*key*,*value*) for the first record. The order of records is arbitrary, except for BTree objects, which ensure records are sorted in alphabetical order of their keys. *b*.first( ) raises KeyError if *b* is empty.

has_key

```
b.has_key(key)
```

11.4 The Python Database API (DBAPI) 2.0

As I mentioned earlier, the Python standard library does not come with an RDBMS interface, but there are many free third-party modules that let your Python programs access specific databases. Such modules mostly follow the Python Database API 2.0 standard, also known as the DBAPI.

At the time of this writing, Python's DBAPI Special Interest Group (SIG) was busy preparing a new version of the DBAPI (possibly to be known as 3.0 when it is ready). Programs written against DBAPI 2.0 should work with minimal or no changes with the future DBAPI 3.0, although 3.0 will no doubt offer further enhancements that future programs will be able to take advantage of.

If your Python program runs only on Windows, you may prefer to access databases by using Microsoft's ADO package through COM. For more information on using Python on Windows, see the book Python Programming on Win32, by Mark Hammond and Andy Robinson (O'Reilly). Since ADO and COM are platform-specific, and this book focuses on cross-platform use of Python, I do not cover ADO nor COM further in this book.

After importing a DBAPI-compliant module, you call the module's connect function with suitable parameters. connect returns an instance of class Connection, which represents a connection to the database. This instance supplies commit and rollback methods to let you deal with transactions, a close method to call as soon as you're done with the database, and a cursor method that returns an instance of class Cursor. This instance supplies the methods and attributes that you'll use for all database operations. A DBAPI-compliant module also supplies exception classes, descriptive attributes, factory functions, and type-description attributes.

11.4.1 Exception Classes

A DBAPI-compliant module supplies exception classes Warning, Error, and several subclasses of Error. Warning indicates such anomalies as data truncation during insertion. Error's subclasses indicate various kinds of errors that your program can encounter when dealing with the database and the DBAPI-compliant module that interfaces to it. Generally, your code uses a statement of the form:

```
try:
    ...
except module.Error, err:
    ...
```

in order to trap all database-related errors that you need to handle without terminating.

11.4.2 Thread Safety

When a DBAPI-compliant module has an attribute threadsafety that is greater than 0, the module is asserting some specific level of thread safety for database interfacing. Rather than relying on this, it's safer and more portable to ensure that a single thread has exclusive access to any given external resource, such as a database, as outlined in Chapter 14.

11.4.3 Parameter Style

A DBAPI-compliant module has an attribute paramstyle that identifies the style of markers to use as placeholders for parameters. You insert such markers in SQL statement strings that you pass to methods of Cursor instances, such as method execute, in order to use runtime-determined parameter values. Say, for example, that you need to fetch the

Chapter 12. Time Operations

A Python program can handle time in several ways. Time intervals are represented by floating-point numbers, in units of seconds (a fraction of a second is the fractional part of the interval). Particular instants in time are expressed in seconds since a reference instant, known as the *epoch*. (Midnight, UTC, of January 1, 1970, is a popular epoch used on both Unix and Windows platforms.) Time instants often also need to be expressed as a mixture of units of measurement (e.g., years, months, days, hours, minutes, and seconds), particularly for I/O purposes.

This chapter covers the time module, which supplies Python's core time-handling functionality. The time module strongly depends on the system C library. The chapter also presents the sched and calendar modules and the essentials of the popular extension module mx.DateTime. mx.DateTime has more uniform behavior across platforms than time, which helps account for its popularity.

Python 2.3 will introduce a new datetime module to manipulate dates and times in other ways. At http://starship.python.net/crew/jbauer/normaldate/, you can download Jeff Bauer's *normalDate.py*, which gains simplicity by dealing only with dates, not with times. Neither of these modules is further covered in this book.

12.1 The time Module

The underlying C library determines the range of dates that the time module can handle. On Unix systems, years 1970 and 2038 are the typical cut-off points, a limitation that mx.DateTime lets you avoid. Time instants are normally specified in UTC (Coordinated Universal Time, once known as GMT, or Greenwich Mean Time). Module time also supports local time zones and Daylight Saving Time (DST), but only to the extent that support is supplied by the underlying C system library.

As an alternative to seconds since the epoch, a time instant can be represented by a tuple of nine integers known as a time-tuple. Items in time-tuples are covered in Table 12-1. All items are integers, and therefore time-tuples cannot keep track of fractions of a second. In Python 2.2 and later, the result of any function in module time that used to return a time-tuple is now of type struct_time. You can still use the result as a tuple, but you can also access the items as read-only attributes *x*.tm_year, *x*.tm_mon, and so on, using the attribute names listed in Table 12-1. Wherever a function used to require a time-tuple argument, you can now pass an instance of struct_time or any other sequence whose items are nine integers in the applicable ranges.

Table 12-1. Tuple form of time representation

Item	**Meaning**	**Field name**	**Range**	**Notes**
0	Year	`tm_year`	1970-2038	Wider on some platforms
1	Month	`tm_mon`	1-12	1 is January; 12 is December
2	Day	`tm_mday`	1-31	
3	Hour	`tm_hour`	0-23	0 is midnight; 12 is noon
4	Minute	`tm_min`	0-59	
5	Second	`tm_sec`	0-61	60 and 61 for leap seconds
6	Weekday	`tm_wday`	0-6	0 is Monday; 6 is Sunday
7	Year day	`tm_yday`	1-366	Day number within the year
8	DST flag	`tm_isdst`	-1 to 1	-1 means library

12.2 The sched Module

The sched module supplies a class that implements an event scheduler. sched supplies a scheduler class.

scheduler

```
class scheduler(timefunc,
delayfunc)
```

An instance *s* of scheduler is initialized with two functions, which *s* then uses for all time-related operations. *timefunc* must be callable without arguments to get the current time instant (in any unit of measure), meaning that you can pass time.time. *delayfunc* must be callable with one argument (a time duration, in the same units *timefunc* returns), and it should delay for about that amount of time, meaning you can pass time.sleep. scheduler also calls *delayfunc* with argument 0 after each event, to give other threads a chance; again, this is compatible with the behavior of time.sleep.

A scheduler instance *s* supplies the following methods.

cancel

```
s.cancel(event_token)
```

Removes an event from *s*'s queue of scheduled events. *event_token* must be the result of a previous call to *s*.enter or *s*.enterabs, and the event must not yet have happened; otherwise cancel raises RuntimeError.

empty

```
s.empty(  )
```

Returns True if *s*'s queue of scheduled events is empty, otherwise False.

enterabs

```
s.enterabs(when,priority,func,
args)
```

Schedules a future event (i.e., a callback to *func*(**args*)) at time *when*. *when* is expressed in the same units of measure used by the time functions of *s*. If several events are scheduled for the same instant, *s* executes them in increasing order of *priority*. enterabs returns an event token *t*, which you may later pass to *s*.cancel to cancel this event.

enter

```
s.enter(delay,priority,func,
args)
```

Like enterabs, except that argument *delay* is a relative time (the difference from the current instant, in the same units of measure), while enterabs's argument *when* is an absolute time (a future instant).

run

12.3 The calendar Module

The calendar module supplies calendar-related functions, including functions to print a text calendar for any given month or year. By default, calendar considers Monday the first day of the week and Sunday the last one. You can change this setting by calling function calendar.setfirstweekday. calendar handles years in the range supported by module time, typically 1970 to 2038. Module calendar supplies the following functions.

calendar

```
calendar(year,w=2,l=1,c=6)
```

Returns a multiline string with a calendar for year *year* formatted into three columns separated by *c* spaces. *w* is the width in characters of each date; each line has length 21**w*+18+2**c*. *l* is the number of lines used for each week.

firstweekday

```
firstweekday(  )
```

Returns the current setting for the weekday that starts each week. By default, when calendar is first imported, this is 0, meaning Monday.

isleap

```
isleap(year)
```

Returns True if *year* is a leap year, otherwise False.

leapdays

```
leapdays(y1,y2)
```

Returns the total number of leap days in the years in range(*y1*,*y2*).

month

```
month(year,month,w=2,l=1)
```

Returns a multiline string with a calendar for month *month* of year *year*, one line per week plus two header lines. *w* is the width in characters of each date; each line has length 7**w*+6. *l* is the number of lines for each week.

monthcalendar

```
monthcalendar(year,month)
```

Returns a list of lists of integers. Each sublist represents a week. Days outside month *month* of year *year* are represented by a placeholder value of 0; days within the given month are represented by their dates, from 1 on up.

monthrange

12.4 The mx.DateTime Module

DateTime is one of the modules in the mx package made available by eGenix GmbH. mx is open source, and at the time of this writing, mx.DateTime has liberal license conditions similar to those of Python itself. mx.DateTime's popularity stems from its functional richness and cross-platform portability. I present only an essential subset of mx.DateTime's rich functionality here; the module comes with detailed documentation about its advanced time and date handling features.

12.4.1 Date and Time Types

Module DateTime supplies several date and time types whose instances are immutable (and therefore suitable as dictionary keys). Type DateTime represents a time instant and includes an absolute date, which is the number of days since an epoch of January 1, year 1 CE, according to the Gregorian calendar (0001-01-01 is day 1), and an absolute time, which is a floating-point number of seconds since midnight. Type DateTimeDelta represents an interval of elapsed time, which is a floating-point number of seconds. Class RelativeDateTime lets you specify dates in relative terms, such as "next Monday" or "first day of next month." DateTime and DateTimeDelta are covered in detail later in this section, but RelativeDateTime is not.

Date and time types supply customized string conversion, invoked via the built-in str or automatically during implicit conversion (e.g., in a print statement). The resulting strings are in standard ISO 8601 formats, such as:

```
 YYYY-MM-DD HH:MM:SS.ss
```

For finer-grained control of string formatting, use method strftime. Function DateTimeFrom constructs DateTime instances from strings. Submodules of module mx.DateTime supply other formatting and parsing functions, using different standards and conventions.

12.4.2 The DateTime Type

Module DateTime supplies factory functions to build instances of type DateTime, which in turn supply methods, attributes, and arithmetic operators.

12.4.2.1 Factory functions for DateTime

Module DateTime supplies many factory functions that produce DateTime instances. Several of these factory functions can also be invoked through synonyms. The most commonly used factory functions are the following.

DateTime, Date, Timestamp

```
DateTime(year,month=1,day=1,
hour=0,minute=0,second=0.0)
```

Creates and returns a DateTime instance representing the given absolute time. Date and Timestamp are synonyms of DateTime. *day* can be less than 0 to denote days counted from the end of the month: -1 is the last day of the month, -2 the next to last day, and so on. For example:

```
 print mx.DateTime.DateTime(2002,12,-1)
# prints: 2002-12-31 00:00:00.00
```

second is a floating-point value and can include an arbitrary fraction of a second.

DateTimeFrom, TimestampFrom

Chapter 13. Controlling Execution

Python directly exposes many of the mechanisms it uses internally. This helps you understand Python at an advanced level, and means you can hook your own code into such documented Python mechanisms and control those mechanisms to some extent. For example, Chapter 7 covered the import statement and the way Python arranges for built-ins to be made implicitly visible. This chapter covers other advanced techniques that Python offers for controlling execution, while Chapter 17 covers execution-control possibilities that apply specifically to the three crucial phases of development: testing, debugging, and profiling.

13.1 Dynamic Execution and the exec Statement

With Python's exec statement, it is possible to execute code that you read, generate, or otherwise obtain during the running of a program. The exec statement dynamically executes a statement or a suite of statements. exec is a simple keyword statement with the following syntax:

```
exec code[ in globals[,locals]]
```

code can be a string, an open file-like object, or a code object. *globals* and *locals* are dictionaries. If both are present, they are the global and local namespaces, respectively, in which *code* executes. If only *globals* is present, exec uses *globals* in the role of both namespaces. If neither *globals* nor *locals* is present, *code* executes in the current scope. Running exec in current scope is not good programming practice, since it can bind, rebind, or unbind any name. To keep things under control, you should use exec only with specific, explicit dictionaries.

13.1.1 Avoiding exec

More generally, use exec only when it's really indispensable. Most often, it is better avoided in favor of more specific mechanisms. For example, a frequently asked question is, "How do I set a variable whose name I just read or constructed?" Strictly speaking, exec lets you do this. For example, if the name of the variable you want to set is in variable *varname*, you might use:

```
exec varname+'=23'
```

Don't do this. An exec statement like this in current scope causes you to lose control of your namespace, leading to bugs that are extremely hard to track and more generally making your program unfathomably difficult to understand. An improvement is to keep the "variables" you need to set, not as variables, but as entries in a dictionary, say *mydict*. You can then use the following variation:

```
exec varname+'=23' in mydict
```

While this is not as terrible as the previous example, it is still a bad idea. The best approach is to keep such "variables" as dictionary entries and not use exec at all to set them. You can just use:

```
mydict[varname] = 23
```

With this approach, your program is clearer, more direct, more elegant, and faster. While there are valid uses of exec, they are extremely rare and they should always use explicit dictionaries.

13.1.2 Restricting Execution

If the global namespace is a dictionary without key '__builtins__', exec implicitly adds that key, referring to module __builtin__ (or to the dictionary thereof), as covered in Chapter 8. If the global namespace dictionary has a key '__builtins__' and the value doesn't refer to the real module __builtin__, *code*'s execution is restricted, as covered in the upcoming section Section 13.2.

13.1.3 Expressions

exec can execute an expression because any expression is also a valid statement (called an expression statement). However, Python ignores the value returned by an expression statement in this case. To evaluate an expression and obtain the expression's value, see built-in function eval, covered in Chapter 8.

13.1.4 Compile and Code Objects

13.2 Restricted Execution

Python code executed dynamically normally suffers no special restrictions. Python's general philosophy is to give the programmer tools and mechanisms that make it easy to write good, safe code, and trust the programmer to use them appropriately. Sometimes, however, trust might not be warranted. When code to execute dynamically comes from an untrusted source, the code itself is untrusted. In such cases it's important to selectively restrict the execution environment so that such code cannot accidentally or maliciously inflict damage. If you never need to execute untrusted code, you can skip this section. However, Python makes it easy to impose appropriate restrictions on untrusted code if you ever do need to execute it.

When the __builtins__ item in the global namespace isn't the standard __builtin__ module (or the latter's dictionary), Python knows the code being run is restricted. Restricted code executes in a sandbox environment, previously prepared by the trusted code, that requests the restricted code's execution. Standard modules rexec and Bastion help you prepare an appropriate sandbox. To ensure that restricted code cannot escape the sandbox, a few crucial internals (e.g., the __dict__ attributes of modules, classes, and instances) are not directly available to restricted code.

There is no special protection against restricted code raising exceptions. On the contrary, Python diagnoses any attempt by restricted code to violate the sandbox restrictions by raising an exception. Therefore, you should generally run restricted code in the try clause of a try/except statement, as covered in Chapter 6. Make sure you catch all exceptions and handle them appropriately if your program needs to keep running in such cases.

There is no built-in protection against untrusted code attempting to inflict damage by consuming large amounts of memory or time (so-called denial-of-service attacks). If you need to ward against such attacks, you can run untrusted code in a separate process. The separate process uses the mechanisms described in this section to restrict the untrusted code's execution, while the main process monitors the separate one and terminates it if and when resource consumption becomes excessive. Processes are covered in Chapter 14. Resource monitoring is currently supported by the standard Python library only on Unix-like platforms (by platform-specific module resource), and this book covers only cross-platform Python.

As a final note, you need to know that there are known, exploitable security weaknesses in the restricted-execution mechanisms, even in the most recent versions of Python. Although restricted execution is better than nothing, at the time of this writing there are no known ways to execute untrusted code that are suitable for security-critical situations.

13.2.1 The rexec Module

The rexec module supplies the RExec class, which you can instantiate to prepare a typical restricted-execution sandbox environment in which to run untrusted code.

RExec

```
class RExec(hooks=None,verbose
=False)
```

Returns an instance of the RExec class, which corresponds to a new restricted-execution environment, also known as a sandbox. *hooks*, if not None, lets you exert fine-grained control on import statements executed in the sandbox. This is an advanced and rarely used functionality, and I do not cover it further in this book. *verbose*, if true, causes additional debugging output to be sent to standard output for many kinds of operations in the sandbox.

13.2.1.1 Methods

13.3 Internal Types

Some of the internal Python objects that I mention in this section are hard to use. Using such objects correctly requires some study of Python's own C (or Java) sources. Such black magic is rarely needed, except to build general-purpose development frameworks and similar wizardly tasks. Once you do understand things in depth, Python empowers you to exert control, if and when you need to. Since Python exposes internal objects to your Python code, you can exert that control by coding in Python, even when a nodding acquaintance with C (or Java) is needed to understand what is going on.

13.3.1 Type Objects

The built-in type named type acts as a factory object, returning objects that are types themselves (type was a built-in function in Python 2.1 and earlier). Type objects don't need to support any special operations except equality comparison and representation as strings. Most type objects are callable, and return new instances of the type when called. In particular, built-in types such as int, float, list, str, tuple, and dict all work this way. The attributes of the types module are the built-in types, each with one or more names. For example, types.DictType and types.DictionaryType both refer to type({ }), also known since Python 2.2 as the built-in type dict. Besides being callable to generate instances, type objects are useful in Python 2.2 and later because you can subclass them, as covered in Chapter 5.

13.3.2 The Code Object Type

As well as by using built-in function compile, you can also get a code object via the func_code attribute of a function or method object. A code object's co_varnames attribute is the tuple of names of local variables, including the formal arguments; the co_argcount attribute is the number of arguments. Code objects are not callable, but you can rebind the func_code attribute of a compatible function object in order to wrap a code object into callable form. Module new supplies a function to create a code object, as well as other functions to create instances, classes, functions, methods, and modules. Such needs are both rare and advanced, and are not covered further in this book.

13.3.3 The frame Type

Function _getframe in module sys returns a frame object from Python's call stack. A frame object has attributes that supply information about the code executing in the frame and the execution state. Modules traceback and inspect help you access and display information, particularly when an exception is being handled. Chapter 17 provides more information about frames and tracebacks.

13.4 Garbage Collection

Python's garbage collection normally proceeds transparently and automatically, but you can choose to exert some direct control. The general principle is that Python collects each object *x* at some time after *x* becomes unreachable, that is, when no chain of references can reach *x* by starting from a local variable of a function that is executing, nor from a global variable of a loaded module. Normally, an object *x* becomes unreachable when there are no references at all to *x*. However, a group of objects can also be unreachable when they reference each other.

Classic Python keeps in each object *x* a count, known as a *reference count*, of how many references to *x* are outstanding. When *x*'s reference count drops to 0, CPython immediately collects *x*. Function getrefcount of module sys accepts any object and returns its reference count (at least 1, since getrefcount itself has a reference to the object it's examining). Other versions of Python, such as Jython, rely on different garbage collection mechanisms, supplied by the platform they run on (e.g., the JVM). Modules gc and weakref therefore apply only to CPython.

When Python garbage-collects *x* and there are no references at all to *x*, Python then finalizes *x* (i.e., calls *x*._ _del_ _( )) and makes the memory that *x* occupied available for other uses. If *x* held any references to other objects, Python removes the references, which in turn may make other objects collectable by leaving them unreachable.

13.4.1 The gc Module

The gc module exposes the functionality of Python's garbage collector. gc deals only with objects that are unreachable in a subtle way, being part of mutual reference loops. In such a loop, each object in the loop refers to others, keeping the reference counts of all objects positive. However, an outside reference no longer exists to the whole set of mutually referencing objects. Therefore, the whole group, also known as cyclic garbage, is unreachable, and therefore garbage collectable. Looking for such cyclic garbage loops takes time, which is why module gc exists.

gc exposes functions you can use to help you keep garbage collection times under control. These functions can sometimes help you track down a memory leak—objects that are not getting collected even though there should be no more references to them—by letting you discover what other objects are in fact holding on to references to them.

collect

```
collect(  )
```

Forces a full cyclic collection run to happen immediately.

disable

```
disable(  )
```

Suspends automatic garbage collection.

enable

```
enable(  )
```

Re-enables automatic garbage collection previously suspended with disable.

garbage

13.5 Termination Functions

The atexit module lets you register termination functions (i.e., functions to be called at program termination, last in, first out). Termination functions are similar to clean-up handlers established by try/finally. However, termination functions are globally registered and called at the end of the whole program, while clean-up handlers are established lexically and called at the end of a specific try clause. Both termination functions and clean-up handlers are called whether the program terminates normally or abnormally, but not when the termination is caused by calling os._exit. Module atexit supplies a single function called register.

register

```
register(func,*args,**kwds)
```

Ensures that func(*args,**kwds) is called at program termination time.

13.6 Site and User Customization

Python provides a specific hook to let each site customize some aspects of Python's behavior at the start of each run. Customization by each single user is not enabled by default, but Python specifies how programs that want to run user-provided code at startup can explicitly request such customization.

13.6.1 The site and sitecustomize Modules

Python loads standard module site just before the main script. If Python is run with option -S, Python does not load site. -S allows faster startup, but saddles the main script with initialization chores. site's tasks are:

1. Putting sys.path in standard form (absolute paths, no duplicates).

2. Interpreting each *.pth* file found in the Python home directory, adding entries to sys.path, and/or importing modules, as each *.pth* file indicates.

3. Adding built-ins used to display information in interactive sessions (quit, exit, copyright, credits, and license).

4. Setting the default Unicode encoding to 'ascii'. site's source code includes two blocks, each guarded by if 0:, one to set the default encoding to be locale dependent, and the other to disable default encoding and decoding between Unicode and plain strings. You may optionally edit *site.py* to select either block.

5. Trying to import sitecustomize (should import sitecustomize raise an ImportError exception, site catches and ignores it). sitecustomize is the module that each site's installation can optionally use for further site-specific customization beyond site's tasks. It is generally best not to edit *site.py*, as any Python upgrade or reinstallation might overwrite your customizations. sitecustomize's main task is often to set the correct default encoding for the site. Western European sites, for example, may choose to call sys.setdefaultencoding('iso-8859-1').

6. After sitecustomize is done, removing from module sys the attribute sys.setdefaultencoding.

Thus, Python's default Unicode encoding can be set only at the start of a run, not changed in midstream during the run. In an emergency, if a specific main script desperately needs to break this guideline and set a different default encoding from that used by all other scripts, you may place the following snippet at the start of the main script:

```
 import sys                              # get the sys module object
reload(sys)                             # restore module sys from disk
sys.setdefaultencoding('iso-8859-15')   # or whatever codec you need
del sys.setdefaultencoding              # ensure against later accidents
```

However, this is not good style. You should refactor your script so that it can accept whatever default encoding the

Chapter 14. Threads and Processes

A thread is a flow of control that shares global state with other threads; all threads appear to execute simultaneously. Threads are not easy to master, but once you do, they may offer a simpler architecture or better performance (faster response, but typically not better throughput) for some problems. This chapter covers the facilities that Python provides for dealing with threads, including the thread, threading, and Queue modules.

A *process* is an instance of a running program. Sometimes you get better results with multiple processes than with threads. The operating system protects processes from one another. Processes that want to communicate must explicitly arrange to do so, via local inter-process communication (IPC). Processes may communicate via files (covered in Chapter 10) or via databases (covered in Chapter 11). In both cases, the general way in which processes communicate using such data storage mechanisms is that one process can write data, and another process can later read that data back. This chapter covers the process-related parts of module os, including simple IPC by means of pipes, and a cross-platform IPC mechanism known as memory-mapped files, supplied to Python programs by module mmap.

Network mechanisms are well suited for IPC, as they work between processes that run on different nodes of a network as well as those that run on the same node. Chapter 19 covers low-level network mechanisms that provide a flexible basis for IPC. Other, higher-level mechanisms, known as distributed computing, such as CORBA, DCOM/COM+, EJB, SOAP, XML-RPC, and .NET, make IPC easier, whether locally or remotely. However, distributed computing is not covered in this book.

14.1 Threads in Python

Python offers multithreading on platforms that support threads, such as Win32, Linux, and most variants of Unix. The Python interpreter does not freely switch threads. Python uses a global interpreter lock (GIL) to ensure that switching between threads happens only between bytecode instructions or when C code deliberately releases the GIL (Python's C code releases the GIL around blocking I/O and sleep operations). An action is said to be *atomic* if it's guaranteed that no thread switching within Python's process occurs between the start and the end of the action. In practice, an operation that looks atomic actually is atomic when executed on an object of a built-in type (augmented assignment on an immutable object, however, is not atomic). However, in general it is not a good idea to rely on atomicity. For example, you never know when you might be dealing with a derived class rather than an object of a built-in type, meaning there might be callbacks to Python code.

Python offers multithreading in two different flavors. An older and lower-level module, thread, offers a bare minimum of functionality, and is not recommended for direct use by your code. The higher-level module threading, built on top of thread, was loosely inspired by Java's threads, and is the recommended tool. The key design issue in multithreading systems is most often how best to coordinate multiple threads. threading therefore supplies several synchronization objects. Module Queue is very useful for thread synchronization as it supplies a synchronized FIFO queue type, which is extremely handy for communication and coordination between threads.

14.2 The thread Module

The only part of the thread module that your code should use directly is the lock objects that module thread supplies. Locks are simple thread-synchronization primitives. Technically, thread's locks are non-reentrant and unowned: they do not keep track of what thread last locked them, so there is no specific owner thread for a lock. A lock is in one of two states, locked or unlocked.

To get a new lock object (in the unlocked state), call the function named allocate_lock without arguments. This function is supplied by both modules thread and threading. A lock object *L* supplies three methods.

acquire

```
L.acquire(wait=True)
```

When *wait* is True, acquire locks *L*. If *L* is already locked, the calling thread suspends and waits until *L* is unlocked, then locks *L*. Even if the calling thread was the one that last locked *L*, it still suspends and waits until another thread releases *L*. When *wait* is False and *L* is unlocked, acquire locks *L* and returns True. When *wait* is False and *L* is locked, acquire does not affect *L*, and returns False.

locked

```
L.locked(  )
```

Returns True if *L* is locked, otherwise False.

release

```
L.release(  )
```

Unlocks *L*, which must be locked. When *L* is locked, any thread may call *L*.release, not just the thread that last locked *L*. When more than one thread is waiting on *L* (i.e., has called *L*.acquire, finding *L* locked, and is now waiting for *L* to be unlocked), release wakes up an arbitrary waiting thread. The thread that calls release is not suspended: it remains ready and continues to execute.

14.3 The Queue Module

The Queue module supplies first-in, first-out (FIFO) queues that support multithread access, with one main class and two exception classes.

Queue

```
class Queue(maxsize=0)
```

Queue is the main class for module Queue and is covered in the next section. When *maxsize* is greater than 0, the new Queue instance *q* is deemed full when *q* has *maxsize* items. A thread inserting an item with the *block* option, when *q* is full, suspends until another thread extracts an item. When *maxsize* is less than or equal to 0, *q* is never considered full, and is limited in size only by available memory, like normal Python containers.

Empty

Empty is the class of the exception that *q*.get(False) raises when *q* is empty.

Full

Full is the class of the exception that *q*.put(*x*,False) raises when *q* is full.

An instance *q* of class Queue supplies the following methods.

empty

```
q.empty(  )
```

Returns True if *q* is empty, otherwise False.

full

```
q.full(  )
```

Returns True if *q* is full, otherwise False.

get, get_nowait

```
q.get(block=True)
```

When *block* is False, get removes and returns an item from *q* if one is available, otherwise get raises Empty. When *block* is True, get removes and returns an item from *q*, suspending the calling thread, if need be, until an item is available. *q*.get_nowait( ) is like *q*.get(False). get removes and returns items in the same order as put inserted them (first in, first out).

14.4 The threading Module

The threading module is built on top of module thread and supplies multithreading functionality in a more usable form. The general approach of threading is similar to that of Java, but locks and conditions are modeled as separate objects (in Java, such functionality is part of every object), and threads cannot be directly controlled from the outside (meaning there are no priorities, groups, destruction, or stopping). All methods of objects supplied by threading are atomic.

threading provides numerous classes for dealing with threads, including Thread, Condition, Event, RLock, and Semaphore. Besides factory functions for the classes detailed in the following sections of this chapter, threading supplies the currentThread factory function.

currentThread

```
currentThread(  )
```

Returns a Thread object for the calling thread. If the calling thread was not created by module threading, currentThread creates and returns a semi-dummy Thread object with limited functionality.

14.4.1 Thread Objects

A Thread object *t* models a thread. You can pass *t*'s main function as an argument when you create *t*, or you can subclass Thread and override the run method (you may also override _ _init_ _, but should not override other methods). *t* is not ready to run when you create it: to make *t* ready (active), call *t*.start( ). Once *t* is active, it terminates when its main function ends, either normally or by propagating an exception. A Thread *t* can be a daemon, meaning that Python can terminate even if *t* is still active, while a normal (non-daemon) thread keeps Python alive until the thread terminates. Class Thread exposes the following constructor and methods.

Thread

```
class Thread(name=None,target
=None,args=(  ),kwargs={  })
```

Always call Thread with named arguments: the number and order of formal arguments may change in the future, but the names of existing arguments are guaranteed to stay. When you instantiate class Thread itself, you should specify *target*: *t*.run calls *target*(**args*,***kwargs*). When you subclass Thread and override run, you normally don't specify *target*. In either case, execution doesn't begin until you call *t*.start( ). *name* is *t*'s name. If *name* is None, Thread generates a unique name for *t*. If a subclass T of Thread overrides _ _init_ _, T._ _init_ _ must call Thread._ _init_ _ on self before any other Thread method.

getName, setName

```
t.getName(  )
t.setName(name)
```

getName returns *t*'s name, and setName rebinds *t*'s name. The *name* string is arbitrary, and a thread's name need not be unique among threads.

isAlive

14.5 Threaded Program Architecture

A threaded program should always arrange for a single thread to deal with any given object or subsystem that is external to the program (such as a file, a database, a GUI, or a network connection). Having multiple threads that deal with the same external object can often cause unpredictable problems.

Whenever your threaded program must deal with some external object, devote a thread to such dealings, using a Queue object from which the external-interfacing thread gets work requests that other threads post. The external-interfacing thread can return results by putting them on one or more other Queue objects. The following example shows how to package this architecture into a general, reusable class, assuming that each unit of work on the external subsystem can be represented by a callable object:

```
 import Threading, Queue
class ExternalInterfacing(Threading.Thread):
    def _ _init_ _(self, externalCallable, **kwds):
        Threading.Thread._ _init_ _(self, **kwds)
        self.setDaemon(1)
        self.externalCallable = externalCallable
        self.workRequestQueue = Queue.Queue(  )
        self.resultQueue = Queue.Queue(  )
        self.start(  )
    def request(self, *args, **kwds):
        "called by other threads as externalCallable would be"
        self.workRequestQueue.put((args,kwds))
        return self.resultQueue.get(  )
    def run(self):
        while 1:
            args, kwds = self.workRequestQueue.get(  )
            self.resultQueue.put(self.externalCallable(*args, **kwds))
```

Once some ExternalInterfacing object *ei* is instantiated, all other threads may now call *ei*.request just like they would call *someExternalCallable* without such a mechanism (with or without arguments as appropriate). The advantage of the ExternalInterfacing mechanism is that all calls upon *someExternalCallable* are now serialized. This means they are performed by just one thread (the thread object bound to *ei*) in some defined sequential order, without overlap, race conditions (hard-to-debug errors that depend on which thread happens to get there first), or other anomalies that might otherwise result.

If several callables need to be serialized together, you can pass the callable as part of the work request, rather than passing it at the initialization of class ExternalInterfacing, for greater generality. The following example shows this more general approach:

```
 import Threading, Queue
class Serializer(Threading.Thread):
    def _ _init_ _(self, **kwds):
        Threading.Thread._ _init_ _(self, **kwds)
        self.setDaemon(1)
        self.workRequestQueue = Queue.Queue(  )
        self.resultQueue = Queue.Queue(  )
        self.start(  )
    def apply(self, callable, *args, **kwds):
        "called by other threads as callable would be"
        self.workRequestQueue.put((callable, args,kwds))
        return self.resultQueue.get(  )
    def run(self):
        while 1:
            callable, args, kwds = self.workRequestQueue.get(  )
            self.resultQueue.put(callable(*args, **kwds))
```

Once a Serializer object *ser* has been instantiated, other threads may call *ser*.apply(*someExternalCallable*) just like

14.6 Process Environment

The operating system supplies each process *P* with an *environment*, which is a set of environment variables whose names are identifiers (most often, by convention, uppercase identifiers) and whose contents are strings. For example, in Chapter 3, we covered environment variables that affect Python's operations. Operating system shells offer various ways to examine and modify the environment, by such means as shell commands and others mentioned in Chapter 3.

The environment of any process *P* is determined when *P* starts. After startup, only *P* itself can change *P*'s environment. Nothing that *P* does affects the environment of *P*'s parent process (the process that started *P*), nor those of child processes previously started from *P* and now running, nor of processes unrelated to *P*. Changes to *P*'s environment affect only *P* itself: the environment is not a means of IPC. Child processes of *P* normally get a copy of *P* 's environment as their starting environment: in this sense, changes to *P*'s environment do affect child processes that *P* starts after such changes.

Module os supplies attribute environ, a mapping that represents the current process's environment. os.environ is initialized from the process environment when Python starts. Changes to os.environ update the current process's environment if the platform supports such updates. Keys and values in os.environ must be strings. On Windows, but not on Unix-like platforms, keys into os.environ are implicitly uppercased. For example, here's how to try to determine what shell or command processor you're running under:

```
import os
shell = os.environ.get('COMSPEC')
if shell is None: shell = os.environ.get('SHELL')
if shell is None: shell = 'an unknown command processor'
print 'Running under', shell
```

If a Python program changes its own environment (e.g., via os.environ['X']='Y'), this does not affect the environment of the shell or command processor that started the program. Like in other cases, changes to a process's environment affect only the process itself, not others.

14.7 Running Other Programs

The os module offers several ways for your program to run other programs. The simplest way to run another program is through function os.system, although this offers no way to control the external program. The os module also provides a number of functions whose names start with exec. These functions offer fine-grained control. A program run by one of the exec functions, however, replaces the current program (i.e., the Python interpreter) in the same process. In practice, therefore, you use the exec functions mostly on platforms that let a process duplicate itself by fork (i.e., Unix-like platforms). Finally, os functions whose names start with spawn and popen offer intermediate simplicity and power: they are cross-platform and not quite as simple as system, but simple and usable enough for most purposes.

The exec and spawn functions run a specified executable file given the executable file's path, arguments to pass to it, and optionally an environment mapping. The system and popen functions execute a command, a string passed to a new instance of the platform's default shell (typically */bin/sh* on Unix, *command.com* or *cmd.exe* on Windows). A command is a more general concept than an executable file, as it can include shell functionality (pipes, redirection, built-in shell commands) using the normal shell syntax specific to the current platform.

execl, execle, execlp, execv, execve, execvp, execvpe

```
execl(path,*args)
execle(path,*args)
execlp(path,*args)
execv(path,args)
execve(path,args,env)
execvp(path,args)
execvpe(path,args,env)
```

These functions run the executable file (program) indicated by string *path*, replacing the current program (i.e., the Python interpreter) in the current process. The distinctions encoded in the function names (after the prefix exec) control three aspects of how the new program is found and run:

- Does *path* have to be a complete path to the program's executable file, or can the function also accept just a name as the *path* argument and search for the executable in several directories, like operating system shells do? execlp, execvp, and execvpe can accept a *path* argument that is just a filename rather than a complete path. In this case, the functions search for an executable file of that name along the directories listed in os.environ['PATH']. The other functions require *path* to be a complete path to the executable file for the new program.

- Are arguments for the new program accepted as a single sequence argument *args* to the function or as separate arguments to the function? Functions whose names start with execv take a single argument *args* that is the sequence of the arguments to use for the new program. Functions whose names start with execl take the new program's arguments as separate arguments (execle, in particular, uses its last argument as the environment for the new program).

- Is the new program's environment accepted as an explicit mapping argument *env* to the function, or is os.environ implicitly used? execle, execve, and execvpe take an argument *env* that is a mapping to be used as the new program's environment (keys and values must be strings), while the other functions use os.environ

14.8 The mmap Module

The mmap module supplies memory-mapped file objects. An mmap object behaves similarly to a plain (not Unicode) string, so you can often pass an mmap object where a plain string is expected. However, there are differences:

- An mmap object does not supply the methods of a string object
- An mmap object is mutable, while string objects are immutable
- An mmap object also corresponds to an open file and behaves polymorphically to a Python file object (as covered in Chapter 10)

An mmap object *m* can be indexed or sliced, yielding plain strings. Since *m* is mutable, you can also assign to an indexing or slicing of *m*. However, when you assign to a slice of *m*, the right-hand side of the assignment statement must be a string of exactly the same length as the slice you're assigning to. Therefore, many of the useful tricks available with list slice assignment (covered in Chapter 4) do not apply to mmap slice assignment.

Module mmap supplies a factory function that is different on Unix-like systems and Windows.

mmap

```
mmap(filedesc,length,tagname
='')    # Windows
mmap(filedesc,length,flags
=MAP_SHARED,
     prot=PROT_READ|PROT_WRITE)
   # Unix
```

Creates and returns an mmap object *m* that maps into memory the first *length* bytes of the file indicated by file descriptor *filedesc*. *filedesc* must normally be a file descriptor opened for both reading and writing (except, on Unix-like platforms, when argument *prot* requests only reading or only writing). File descriptors are covered in Section 10.2.8. To get an mmap object *m* that refers to a Python file object *f*, use *m*=mmap.mmap(*f*.fileno( ),*length*).

On Windows only, you can pass a string *tagname* to give an explicit tag name for the memory mapping. This tag name lets you have several memory mappings on the same file, but this functionality is rarely necessary. Calling mmap with only two arguments has the advantage of keeping your code portable between Windows and Unix-like platforms. On Windows, all memory mappings are readable and writable and shared between processes, so that all processes with a memory mapping on a file can see changes made by each such process.

On Unix-like platforms only, you can pass mmap.MAP_PRIVATE as the *flags* argument to get a mapping that is private to your process and copy-on-write. mmap.MAP_SHARED, the default, gets a mapping that is shared with other processes, so that all processes mapping the file can see changes made by one process (same as on Windows). You can pass mmap.PROT_READ as the *prot* argument to get a mapping that you can only read, not write. Passing mmap.PROT_WRITE gets a mapping that you can only write, not read. The bitwise-OR

Chapter 15. Numeric Processing

In Python, you can perform numeric computations with operators (as covered in Chapter 4) and built-in functions (as covered in Chapter 8). Python also provides the math, cmath, operator, and random modules, which support additional numeric computation functionality, as documented in this chapter.

You can represent arrays in Python with lists and tuples (covered in Chapter 4), as well as with the array standard library module, which is covered in this chapter. You can also build advanced array manipulation functions with loops, list comprehensions, iterators, generators, and built-ins such as map, reduce, and filter, but such functions can be complicated and slow. Therefore, when you process large arrays of numbers in these ways, your program's performance can be below your machine's full potential.

The Numeric package addresses these issues, providing high-performance support for multidimensional arrays (matrices) and advanced mathematical operations, such as linear algebra and Fourier transforms. Numeric does not come with standard Python distributions, but you can freely download it at http://sourceforge.net/projects/numpy, either as source code (which is easy to build and install on many platforms) or as a prebuilt self-installing *.exe* file for Windows. Visit http://www.pfdubois.com/numpy/ for an extensive tutorial and other resources, such as a mailing list about Numeric. Note that the Numeric package is not just for numeric processing. Much of Numeric is about multidimensional arrays and advanced array handling that you can use for any Python sequence.

Numeric is a large, rich package. For full understanding, study the tutorial, work through the examples, and experiment interactively. This chapter presents a reference to an essential subset of Numeric on the assumption that you already have some grasp of array manipulation and numeric computing issues. If you are unfamiliar with this subject, the Numeric tutorial can help.

15.1 The math and cmath Modules

The math module supplies mathematical functions on floating-point numbers, while the cmath module supplies equivalent functions on complex numbers. For example, math.sqrt(-1) raises an exception, but cmath.sqrt(-1) returns 1j.

Each module also exposes two attributes of type float bound to the values of fundamental mathematical constants, pi and e.

acos *math and cmath*

```
acos(x)
```

Returns the arccosine of *x* in radians.

acosh *cmath only*

```
acosh(x)
```

Returns the arc hyperbolic cosine of *x* in radians.

asin *math and cmath*

```
asin(x)
```

Returns the arcsine of *x* in radians.

asinh *cmath only*

```
asinh(x)
```

Returns the arc hyperbolic sine of *x* in radians.

atan *math and cmath*

```
atan(x)
```

Returns the arctangent of *x* in radians.

atanh *cmath only*

```
atanh(x)
```

Returns the arc hyperbolic tangent of *x* in radians.

atan2 *math only*

15.2 The operator Module

The operator module supplies functions that are equivalent to Python's operators. These functions are handy for use with map and reduce, and in other cases where callables must be stored, passed as arguments, or returned as function results. The functions in operator have the same names as the corresponding special methods (covered in Chapter 5). Each function is available with two names, with and without the leading and trailing double underscores (e.g., both operator.add(*a*,*b*) and operator._ _add_ _(*a*,*b*) return *a*+*b*). Table 15-1 lists the functions supplied by operator.

Table 15-1. Functions supplied by operator

Method	**Signature**	**Behaves like**
`abs`	`abs(`*a*`)`	abs(*a*)
`add`	`add(`*a*`,`*b*`)`	*a*+*b*
`and_`	`and_(`*a*`,`*b*`)`	*a*&*b*
`concat`	`concat(`*a*`,`*b*`)`	*a*+*b*
`contains`	`contains(`*a*`,`*b*`)`	*b* in *a*
`countOf`	`countOf(`*a*`,`*b*`)`	*a*.count(*b*)
`delitem`	`delitem(`*a*`,`*b*`)`	del *a*[*b*]
`delslice`	`delslice(`*a*`,`*b*`,`*c*`)`	del *a*[*b*:*c*]
`div`	`div(`*a*`,`*b*`)`	*a*/*b*
`getitem`	`getitem(`*a*`,`*b*`)`	*a*[*b*]
`getslice`	`getslice(`*a*`,`*b*`,`*c*`)`	*a*[*b*:*c*]
`indexOf`	`indexOf(`*a*`,`*b*`)`	*a*.index(*b*)
[illegible]	invert(*a*), inv(*a*)	[illegible]

15.3 The random Module

The random module generates pseudo-random numbers with various distributions. The underlying uniform pseudo-random generator uses the Whichmann-Hill algorithm, with a period of length 6,953,607,871,644. The resulting pseudo-random numbers, while quite good, are not of cryptographic quality. If you want physically generated random numbers rather than algorithmically generated pseudo-random numbers, you may use */dev/random* or */dev/urandom* on platforms that support such pseudo-devices (such as recent Linux releases). For an alternative, see http://www.fourmilab.ch/hotbits.

All functions of module random are methods of a hidden instance of class random.Random. You can instantiate Random explicitly to get multiple generators that do not share state. Explicit instantiation is advisable if you require random numbers in multiple threads (threads are covered in Chapter 14). This section documents the most frequently used functions exposed by module random.

choice

```
choice(seq)
```

Returns a random item from non-empty sequence *seq*.

getstate

```
getstate(  )
```

Returns an object *S* that represents the current state of the generator. You can later pass *S* to function setstate in order to restore the generator's state.

jumpahead

```
jumpahead(n)
```

Advances the generator state as if *n* random numbers had been generated. Computing the new state is faster than generating *n* random numbers would be.

random

```
random(  )
```

Returns a random floating-point number *r* from a uniform distribution, such that 0<=*r*<1.

randrange

```
randrange([start,]stop[,step])
```

Like choice(range(*start*,*stop*,*step*)), but faster, since randrange does not need to build the list that range would create.

seed

15.4 The array Module

The array module supplies a type, also called array, whose instances are mutable sequences, like lists. An array *a* is a one-dimensional sequence whose items can be only characters, or only numbers of one specific numeric type that is fixed when *a* is created.

The extension module Numeric, covered later in this chapter, also supplies a type called array that is far more powerful than array.array. For advanced array operations and multidimensional arrays, I recommend Numeric even if your array elements are not numbers.

array.array is a simple type, whose main advantage is that, compared to a list, it can save memory to hold objects all of the same (numeric or character) type. An array object *a* has a one-character read-only attribute *a*.typecode, set when *a* is created, that gives the type of *a*'s items. Table 15-2 shows the possible type codes for array.

Table 15-2. Type codes for the array module

Type code	**C type**	**Python type**	**Minimum size**
`'c'`	`char`	str (length 1)	1 byte
`'b'`	`char`	`int`	1 byte
`'B'`	`unsigned char`	`int`	1 byte
`'h'`	`short`	`int`	2 bytes
`'H'`	`unsigned short`	`int`	2 bytes
`'i'`	`int`	`int`	2 bytes
`'I'`	`unsigned`	`long`	2 bytes
`'l'`	`long`	`int`	4 bytes
`'L'`	`unsigned long`	`long`	4 bytes
`'f'`	`float`	`float`	4 bytes
`'d'`	`double`	`float`	8 bytes

15.5 The Numeric Package

The main module in the Numeric package is the Numeric module, which provides the array object type, a set of functions that manipulate these objects, and universal functions that operate on arrays and other sequences. The Numeric package also supports a variety of optional modules for things like linear algebra, random numbers, masked arrays, and Fast Fourier Transforms.

Numeric is one of the rare Python packages often used with the idiom from Numeric import *. You can also use import Numeric and qualify each name by preceding it with Numeric. However, if you need many of the package's names, importing all the names at once is handy. Another popular alternative is to import Numeric with a shorter name (e.g., import Numeric as N) and qualify each name by preceding it with N.

Although quite solid and stable, Numeric is under continuous development, with functionality being added and limitations removed. This chapter describes specifically Numeric Version 21.3, the latest released version at the time of this writing. A successor to Numeric, named numarray, is being developed by the Numeric community, and is not quite ready for production use yet. numarray is not totally compatible with Numeric, but shares most of Numeric's functionality and enriches it further. Information on numarray is available at http://stsdas.stsci.edu/numarray/.

15.6 Array Objects

Numeric provides an array type that represents a grid of items. An array object *a* has a specified number of dimensions, known as its rank, up to some arbitrarily high limit (normally 40, when Numeric is built with default options). A scalar (i.e., a single number) has rank 0, a vector has rank 1, a matrix has rank 2, and so forth.

15.6.1 Type Codes

The values that occupy cells in the grid of an array object, known as the elements of the array, are homogeneous, meaning they are all of the same type, and all element values are stored within one memory area. This contrasts with a list or tuple, where the items may be of different types and each is stored as a separate Python object. This means a Numeric array occupies far less memory than a Python list or tuple with the same number of items. The type of *a*'s elements is encoded as *a*'s type code, a one-character string, as shown in Table 15-3. Factory functions that build array instances, covered in Section 15.6.6 later in this chapter, take a *typecode* argument that is one of the values in Table 15-3.

Table 15-3. Type codes for Numeric arrays

Type code	**C type**	**Python type**	**Synonym**
`'c'`	`char`	str (length 1)	`Character`
`'b'`	`unsigned char`	`int`	`UnsignedInt8`
`'1'`	`signed char`	`int`	`Int8`
`'s'`	`short`	`int`	`Int16`
`'i'`	`int`	`int`	`Int32`
`'l'`	`long`	`int`	`Int`
`'f'`	`float`	`float`	`Float32`
`'F'`	two floats	`complex`	`Complex32`
`'d'`	`double`	`float`	`Float`
`'D'`	two doubles	`complex`	`Complex`
`'O'`	`PyObject*`	`any`	`PyObject`

Numeric supplies readable attribute names for each type code, as shown in the last column of Table 15-3. Numeric also supplies, on all platforms, the names Int0, Float0, Float8, Float16, Float64, Complex0, Complex8, Complex16, and Complex64. In each case, the name refers to the smallest type of the requested kind with at least that many bits. For example, Float8 is the smallest floating-point type of at least 8 bits (generally the same as Float32, but some platforms may provide very small floating-point types), while Complex0 is the smallest complex type. On some platforms, but not all, Numeric also supplies the names Int64, Int128, Float128, and Complex128, with similar meanings. These names are not supplied on all platforms because not all platforms provide numbers with that many bits. The next release of Numeric will also support unsigned integer types.

A type code of 'O' indicates that elements are references to Python objects. In this case, elements can be of different

15.7 Universal Functions (ufuncs)

Numeric supplies named functions with the same semantics as Python's arithmetic, comparison, and bitwise operators. Similar semantics (element-wise operation, broadcasting, coercion) are also available with other mathematical functions, both binary and unary, that Numeric supplies. For example, Numeric supplies typical mathematical functions similar to those supplied by built-in module math, such as sin, cos, log, and exp.

These functions are objects of type ufunc (which stands for universal function) and share several traits in addition to those they have in common with array operators. Every ufunc instance *u* is callable, is applicable to sequences as well as to arrays, and lets you specify an optional *output* argument. If *u* is binary (i.e., if *u* accepts two operand arguments), *u* also has four callable attributes, named *u*.accumulate, *u*.outer, *u*.reduce, and *u*.reduceat. The ufunc objects supplied by Numeric apply only to arrays with numeric type codes (i.e., not to arrays with type code 'O' or 'c').

Any ufunc *u* applies to sequences, not just to arrays. When you start with a list *L*, it's faster to call *u* directly on *L* rather than to convert *L* to an array. *u*'s return value is an array *a*; you can perform further computation, if any, on *a*, and then, if you need a list result, you can convert the resulting array to a list by calling its method tolist. For example, say you must compute the logarithm of each item of a list and return another list. On my system, with N set to 2222 and using python -O, a list comprehension such as:

```
def logsupto(N):
    return [math.log(x) for x in range(2,N)]
```

takes about 5.6 milliseconds. Using Python's built-in map:

```
def logsupto(N):
    return map(math.log, range(2,N))
```

takes around half the time, 2.8 milliseconds. Using Numeric's ufunc named log:

```
def logsupto(N):
    return Numeric.log(range(2,N)).tolist(  )
```

reduces the time to about 2.0 milliseconds. Taking some care to exploit the *output* argument to the log ufunc:

```
def logsupto(N):
    temp = Numeric.arange(2, N, typecode=Numeric.Float)
    Numeric.log(temp, temp)
    return temp.tolist(  )
```

further reduces the time, down to just 0.9 milliseconds. The ability to accelerate such simple but massive computations (here by about 6 times) with so little effort is a good part of the attraction of Numeric, and particularly of Numeric's ufunc objects.

15.7.1 The Optional output Argument

Any ufunc *u* accepts an optional last argument *output* that specifies an output array. If supplied, *output* must be an array or array slice of the right shape and type for *u*'s results (i.e., no coercion, no broadcasting). *u* stores results in *output* and does not create a new array. *output* can be the same as an input array argument *a* of *u*. Indeed, *output* is normally specified in order to substitute common idioms such as *a*=*u*(*a*,*b*) with faster equivalents such as *u*(*a*,*b*,*a*). However, *output* cannot share data with *a* without being *a* (i.e., *output* can't be a different view of some or all of *a*'s data). If you pass such a disallowed *output* argument, Numeric is normally unable to diagnose your error and raise an exception, so instead you get wrong results.

15.8 Optional Numeric Modules

Many other modules are built on top of Numeric or cooperate with it. You can download some of them from the same URL as Numeric (http://sourceforge.net/projects/numpy). Some of these extra modules may already be included in the package you have downloaded. Documentation for the modules is also part of the documentation for Numeric. A rich library of scientific tools that work well with Numeric is SciPy, available at http://www.scipy.org. I highly recommend it if you are using Python for scientific or engineering computing.

Here are some key optional Numeric modules:

MLab

MLab supplies many Python functions written on top of Numeric. MLab's functions are similar in name and operation to functions supplied by the product Matlab.

FFT

FFT supplies Python-callable Fast Fourier Transforms (FFTs) of data held in Numeric arrays. FFT can wrap either the well-known *FFTPACK* Fortran-coded library or the compatible C-coded *fftpack* library.

LinearAlgebra

LinearAlgebra supplies Python-callable functions, operating on data held in Numeric arrays, that wrap either the well-known *LAPACK* Fortran-coded library or the compatible C-coded *lapack_lite* library. LinearAlgebra lets you invert matrices, solve linear systems, compute eigenvalues and eigenvectors, perform singular value decomposition, and least-squares-solve overdetermined linear systems.

RandomArray

RandomArray supplies fast, high-quality pseudo-random number generators, using various random distributions, that work with Numeric arrays.

MA

MA supports masked arrays (i.e., arrays that can have missing or invalid values). MA supplies a large subset of Numeric's functionality, albeit sometimes at reduced speed. The extra functionality of MA is the ability to associate to each array an optional mask, an auxiliary array of False and True, where True indicates array elements that are missing, unknown, or invalid. Computations propagate masks, and you can turn masked arrays into plain Numeric ones by using a fill-in value for invalid elements. MA is widely applicable because experimental data quite often has missing or inapplicable elements. Furthermore, when you need to extend or specialize some aspect of Numeric's behavior for your application's purposes, it often turns out to be simplest and most effective to start with MA's sources rather than with Numeric's. The latter are often quite hard to understand and modify, due to the extreme degree of optimization applied to them over the years.

Chapter 16. Tkinter GUIs

Most professional applications interact with users through a graphical user interface (GUI). A GUI is normally programmed through a toolkit, which is a library that implements *controls* (also known as *widgets*) that are visible objects such as buttons, labels, text entry fields, and menus. A GUI toolkit lets you compose controls into a coherent whole, display them on-screen, and interact with the user, receiving input via such devices as the keyboard and mouse.

Python gives you a choice among many GUI toolkits. Some are platform-specific, but most are cross-platform to different degrees, supporting at least Windows and Unix-like platforms, and often the Macintosh as well. Check http://phaseit.net/claird/comp.lang.python/python_GUI.html for a list of dozens of GUI toolkits available for Python. One package, anygui (http://anygui.org), lets you program simple GUIs to one common programming interface and deploy them with any of a variety of backends.

The most widespread Python GUI toolkit is Tkinter. Tkinter is an object-oriented Python wrapper around the cross-platform toolkit Tk, which is also used with other scripting languages such as Tcl (for which it was originally developed) and Perl. Tkinter, like the underlying Tcl/Tk, runs on Windows, Macintosh, and Unix-like platforms. Tkinter itself comes with standard Python distributions. On Windows, the standard Python distribution also includes the Tcl/Tk components needed to run Tkinter. On other platforms, you must obtain and install Tcl/Tk separately.

This chapter covers an essential subset of Tkinter, sufficient to build simple graphical frontends for Python applications. A richer introduction is available at http://www.pythonware.com/library/tkinter/introduction/.

16.1 Tkinter Fundamentals

The Tkinter module makes it easy to build simple GUI applications. You simply import Tkinter, create, configure, and position the widgets you want, and then enter the Tkinter main loop. Your application becomes *event-driven*, which means that the user interacts with the widgets, causing events, and your application responds via the functions you installed as handlers for these events.

The following example shows a simple application that exhibits this general structure:

```
 import sys, Tkinter
Tkinter.Label(text="Welcome!").pack(   )
Tkinter.Button(text="Exit", command=sys.exit).pack(   )
Tkinter.mainloop(   )
```

The calls to Label and Button create the respective widgets and return them as results. Since we specify no parent windows, Tkinter puts the widgets directly in the application's main window. The named arguments specify each widget's configuration. In this simple case, we don't need to bind variables to the widgets. We just call the pack method on each widget, handing control of the widget's geometry to a layout manager object known as the packer. A *layout manager* is an invisible component whose job is to position widgets within other widgets (known as *container* or *parent* widgets), handling geometrical layout issues. The previous example passes no arguments to control the packer's operation, so therefore the packer operates in a default way.

When the user clicks on the button, the command callable of the Button widget executes without arguments. The example passes function sys.exit as the argument named command when it creates the Button. Therefore, when the user clicks on the button, sys.exit( ) executes and terminates the application (as covered in Chapter 8).

After creating and packing the widgets, the example calls Tkinter's mainloop function, and thus enters the Tkinter main loop and becomes event-driven. Since the only event for which the example installs a handler is a click on the button, nothing happens from the application's viewpoint until the user clicks the button. Meanwhile, however, the Tkinter toolkit responds in the expected way to other user actions, such as moving the Tkinter window, covering and uncovering the window, and so on. When the user resizes the window, the packer layout manager works to update the widgets' geometry. In this example, the widgets remain centered, close to the upper edge of the window, with the label above the button.

All strings going to or coming from Tkinter are Unicode strings, so be sure to review Section 9.6 in Chapter 9 if you need to show, or accept as input, characters outside of the ASCII encoding (you may then need to use some other appropriate codec).

Note that all the scripts in this chapter are meant to be run standalone (i.e., from a command line or in a platform-dependent way, such as by double clicking on a script's icon). Running a GUI script from inside another program that has its own GUI, such as a Python integrated development environment (e.g., IDLE or PythonWin), can cause various anomalies. This can be a particular problem when the GUI script attempts to terminate (and thus close down the GUI), since the script's GUI and the other program's GUI may interfere with each other.

Note also that this chapter refers to several all-uppercase, multi-letter identifiers (e.g., LEFT, RAISED, ACTIVE). All these identifiers are constant attributes of module Tkinter, used for a wide variety of purposes. If your code uses from Tkinter import *, you can then use the identifiers directly. If your code uses import Tkinter instead, you need to qualify those identifiers, just like all others you import from Tkinter, by preceding them with 'Tkinter.'. Tkinter is one of the rare Python modules designed to support from Tkinter import *, but of course you may choose to use import Tkinter anyway, sacrificing some convenience and brevity in favor of greater clarity. A good compromise between convenience and clarity is often to import Tkinter with a shorter name (e.g., import Tkinter as Tk).

16.2 Widget Fundamentals

The Tkinter module supplies many kinds of widgets, and most of them have several things in common. All widgets are instances of classes that inherit from class Widget. Class Widget itself is *abstract*; that is, you never instantiate Widget itself. You only instantiate concrete subclasses corresponding to specific kinds of widgets. Class Widget's functionality is common to all the widgets you instantiate.

To instantiate any kind of widget, call the widget's class. The first argument is the parent window of the widget, also known as the widget's *master*. If you omit this positional argument, the widget's master is the application's main window. All other arguments are in named form, *option*=*value*. You can also set or change options on an existing widget *w* by calling *w*.config(*option*=*value*). You can get an option of *w* by calling *w*.cget('*option*'), which returns the option's value. Each widget *w* is a mapping, so you can also get an option as *w*['*option*'] and set or change it with *w*['*option*']=*value*.

16.2.1 Common Widget Options

Many widgets accept some common options. Some options affect a widget's colors, others affect lengths (normally in pixels), and there are various other kinds. This section details the most commonly used options.

16.2.1.1 Color options

Tkinter represents colors with strings. The string can be a color name, such as 'red' or 'orange', or it may be of the form '#*RRGGBB*', where each of *R*, *G*, and *B* is a hexadecimal digit, to represent a color by the values of red, green, and blue components on a scale of 0 to 255. Don't worry; if your screen can't display millions of different colors, as implied by this scheme; Tkinter maps any requested color to the closest color that your screen can display. The common color options are:
activebackground

Background color for the widget when the widget is *active*, meaning that the mouse is over the widget and clicking on it makes something happen
activeforeground

Foreground color for the widget when the widget is active
background (also bg)

Background color for the widget
disabledforeground

Foreground color for the widget when the widget is *disabled*, meaning that clicking on the widget is ignored
foreground (also fg)

Foreground color for the widget
highlightbackground

Background color of the highlight region when the widget has focus
highlightcolor

Foreground color of the highlight region when the widget has focus
selectbackground

Background color for the selected items of the widget, for widgets that have selectable items, such as Listbox

16.3 Commonly Used Simple Widgets

The Tkinter module provides a number of simple widgets that cover most needs of basic GUI applications. This section documents the Button, Checkbutton, Entry, Label, Listbox, Radiobutton, Scale, and Scrollbar widgets.

16.3.1 Button

Class Button implements a *pushbutton*, which the user clicks to execute an action. Instantiate Button with option text=*somestring* to let the button show text, or image=*imageobject* to let the button show an image. You normally use option command=*callable* to have *callable* execute without arguments when the user clicks the button. *callable* can be a function, a bound method of an object, an instance of a class with a __call__ method, or a lambda.

Besides methods common to all widgets, an instance *b* of class Button supplies two button-specific methods.

flash

```
b.flash(  )
```

Draws the user's attention to button *b* by redrawing *b* a few times, alternatively in normal and active states.

invoke

```
b.invoke(  )
```

Calls without arguments the callable object that is *b*'s command option, just like *b*.cget('command')( ). This can be handy when, within some other action, you want the program to act just as if the button had been clicked.

16.3.2 Checkbutton

Class Checkbutton implements a *checkbox*, which is a little box, optionally displaying a checkmark, that the user clicks to toggle on or off. You normally instantiate Checkbutton with exactly one of the two options text=*somestring*, to label the box with text, or image=*imageobject*, to label the box with an image. Optionally, use option command= *callable* to have *callable* execute without arguments when the user clicks the box. *callable* can be a function, a bound method of an object, an instance of a class with a __call__ method, or a lambda.

An instance *c* of Checkbutton must be associated with a Tkinter variable object *v*, using configuration option variable=*v* of *c*. Normally, *v* is an instance of IntVar, and *v*'s value is 0 when the box is unchecked, and 1 when the box is checked. The value of *v* changes when the box is checked or unchecked (either by the user clicking on it, or by your code calling *c*'s methods deselect, select, toggle). Vice versa, when the value of *v* changes, *c* shows or hides the checkmark as appropriate.

Besides methods common to all widgets, an instance *c* of class Checkbutton supplies five checkbox-specific methods.

deselect

```
c.deselect(  )
```

16.4 Container Widgets

The Tkinter module supplies widgets whose purpose is to contain other widgets. A Frame instance does nothing more than act as a container. A Toplevel instance (including Tkinter's root window, also known as the application's main window) is a top-level window, so your window manager interacts with it (typically by supplying suitable decoration and handling certain requests). To ensure that a widget *parent*, which must be a Frame or Toplevel instance, is the parent (also known as master) of another widget *child*, pass *parent* as the first parameter when you instantiate *child*.

16.4.1 Frame

Class Frame represents a rectangular area of the screen contained in other frames or top-level windows. Frame's only purpose is to contain other widgets. Option borderwidth defaults to 0, so an instance of Frame normally displays no border. You can configure the option with borderwidth=1 if you want the frame border's outline to be visible.

16.4.2 Toplevel

Class Toplevel represents a rectangular area of the screen that is a top-level window and therefore receives decoration from whatever window manager handles your screen. Each instance of Toplevel can interact with the window manager and can contain other widgets. Every program using Tkinter has at least one top-level window, known as the root window. You can instantiate Tkinter's root window explicitly using *root*=Tkinter.Tk( ); otherwise Tkinter instantiates its root window implicitly as and when first needed. If you want to have more than one top-level window, first instantiate the main one with *root*=Tkinter.Tk( ). Later in your program, you can instantiate other top-level windows as needed, with calls such as *another_toplevel*=Tkinter.Toplevel( ).

An instance *T* of class Toplevel supplies many methods enabling interaction with the window manager. Many are platform-specific, relevant only with some window managers for the X Windowing System (used mostly on Unix and Unix-like systems). The cross-platform methods used most often are as follows.

deiconify

```
T.deiconify(  )
```

Makes *T* display normally, even if previously *T* was iconic or invisible.

geometry

```
T.geometry([geometry_string])
```

T.geometry( ), without arguments, returns a string encoding *T*'s size and position: *width*x*height*+*x_offset*+*y_offset*, with *width*, *height*, *x_offset*, and *y_offset* being the decimal forms of the corresponding numbers of pixels. *T* .geometry(*S*), with one argument *S* (a string of the same form), sets *T*'s size and position according to *S*.

iconify

```
T.deiconify(  )
```

Makes *T* display as an icon (in Windows, as a button in the taskbar).

16.5 Menus

Class Menu implements all kinds of menus: menubars of top-level windows, submenus, and pop-up menus. To use a Menu instance *m* as the menubar for a top-level window *w*, set *w*'s configuration option menu=*m*. To use *m* as a submenu of a Menu instance *x*, call *x*.add_cascade with a named argument menu=*m*. To use *m* as a pop-up menu, call method *m*.post.

Besides configuration options covered in Section 16.2.1 earlier in this chapter, a Menu instance *m* supports option postcommand=*callable*. Tkinter calls *callable* without arguments each time it is about to display *m* (whether because of a call to *m*.post or because of user actions). You can use this option to update a dynamic menu just in time when necessary.

By default, a Tkinter menu shows a tear-off entry (a dashed line before other entries), which lets the user get a copy of the menu in a separate Toplevel window. Since such tear-offs are not part of user interface standards on popular platforms, you may want to disable tear-off functionality by using configuration option tearoff=0 for the menu.

16.5.1 Menu-Specific Methods

Besides methods common to all widgets, an instance *m* of class Menu supplies several menu-specific methods.

add, add_cascade, add_checkbutton, add_command, add_radiobutton, add_separator

```
m.add(entry_kind, **
entry_options)
```

Adds after *m*'s existing entries a new entry whose kind is the string *entry_kind*, which is one of the strings 'cascade', 'checkbutton', 'command', 'radiobutton', or 'separator'. Section 16.5.2 later in this chapter covers entry kinds and options.

Methods whose names start with add_ work just like method add, but they accept no positional argument; what kind of entry each method adds is implied by the method's name.

delete

```
m.delete(i[,j])
```

m.delete(*i*) removes *m*'s *i* entry. *m*.delete(*i*,*j*) removes *m*'s entries from the *i* one to the *j* one, included. The first entry has index 0.

entryconfigure, entryconfig

```
m.entryconfigure(i, **
entry_options)
```

Changes entry options for *m*'s *i* entry. entryconfig is an exact synonym.

insert, insert_cascade

16.6 The Text Widget

Class Text implements a powerful multiline text editor, able to display images and embedded widgets as well as text in one or more fonts and colors. An instance *t* of Text supports many ways to refer to specific points in *t*'s contents. *t* supplies methods and configuration options allowing fine-grained control of operations, content, and rendering. This section covers a large, frequently used subset of this vast functionality. In some very simple cases, you can get by with just three Text-specific idioms:

```
t.delete('1.0', END)           # clear the widget's contents
t.insert(END, astring)         # append astring to the widget's contents
somestring = t.get('1.0', END)   # get the widget's contents as a string
```

END is an index on any Text instance *t*, indicating the end of *t*'s text. '1.0' is also an index, indicating the start of *t*'s text (first line, first column). For more about indices, see Section 16.6.5 later in this chapter.

16.6.1 Text Widget Methods

An instance *t* of class Text supplies many methods. Methods dealing with marks and tags are covered in later sections. Many methods accept one or two indices into *t*'s contents. The most frequently used methods are the following.

delete

```
t.delete(i[,j])
```

t.delete(*i*) removes *t*'s character at index *i*. *t*.delete(*i*,*j*) removes all characters from index *i* to index *j*, included.

get

```
t.get(i[,j])
```

t.get(*i*) returns *t*'s character at index *i*. *t*.get(*i*,*j*) returns a string made up of all characters from index *i* to index *j*, included.

image_create

```
t.image_create(i,**
window_options)
```

Inserts an embedded image in *t*'s contents at index *i*. Call image_create with option image=*e*, where *e* is a Tkinter image object, as covered in Section 16.2.4 earlier in this chapter.

insert

```
t.insert(i,s[,tags])
```

Inserts string *s* in *t*'s contents at index *i*. *tags*, if supplied, is a sequence of strings to attach as tags to the new text, as covered in Section 16.6.4 later in this chapter.

search

16.7 The Canvas Widget

Class Canvas is a powerful, flexible widget used for many purposes, including plotting and, in particular, building custom widgets. Building custom widgets is an advanced topic, and I do not cover it further in this book. This section covers only a subset of Canvas functionality used for the simplest kind of plotting.

Coordinates within a Canvas instance *c* are in pixels, with the origin at the upper left corner of *c* and positive coordinates growing rightward and downward. There are advanced methods that let you change *c*'s coordinate system, but I do not cover them in this book.

What you draw on a Canvas instance *c* are canvas items, which can be lines, polygons, Tkinter images, arcs, ovals, texts, and others. Each item has an *item handle* by which you can refer to the item. You can also assign symbolic names called tags to sets of canvas items (the sets of items with different tags can overlap). ALL is a predefined tag that applies to all items; CURRENT is a predefined tag that applies to the item under the mouse pointer.

Tags on a Canvas instance are different from tags on a Text instance. The canvas tags are nothing more than sets of items with no independent existence. When you perform any operation, passing a Canvas tag as the item identifier, the operation occurs on those items that are in the tag's current set. It makes no difference if items are later removed from or added to that tag's set.

You create a canvas item by calling on *c* a method with a name of the form create_*kindofitem*, which returns the new item's handle. Methods itemcget and itemconfig of *c* let you get and change items' options.

16.7.1 Canvas Methods on Items

A Canvas instance *c* supplies methods that you can call on items. The *item* argument can be an item's handle, as returned for example by *c*.create_line, or a tag, meaning all items in that tag's set (or no items at all, if the tag's set is currently empty), unless otherwise indicated in the method's description.

bbox

```
c.bbox(item)
```

Returns an approximate bounding box for *item*, a tuple of four integers: the pixel coordinates of minimum x, minimum y, maximum x, maximum y, in this order. For example, *c*.bbox(ALL) returns the minimum and maximum x and y coordinates of all items on *c*. When *c* has no items at all, *c*.bbox(ALL) returns None.

coords

```
c.coords(item,*coordinates)
```

Changes the coordinates for *item*. Operates on just one item. If *item* is a tag, coords operates on an arbitrary one of the items currently in the tag's set. If *item* is a tag with an empty set, coords is an innocuous no-operation.

delete

```
c.delete(item)
```

16.8 Geometry Management

In all the examples so far, we have made each widget visible by calling method pack on the widget. This is representative of real-life Tkinter usage. However, two other layout managers exist and are sometimes useful. This section covers all three layout managers provided by the Tkinter module.

Never mix geometry managers for the same container widget: all children of each given container widget must be handled by the same geometry manager, or very strange effects (including Tkinter going into infinite loops) may result.

16.8.1 The Packer

Calling method pack on a widget delegates widget geometry management to a simple and flexible layout manager component called the Packer. The Packer sizes and positions each widget within a container (parent) widget, according to each widget's space needs (including options padx and pady). Each widget *w* supplies the following Packer-related methods.

pack

```
w.pack(**pack_options)
```

Delegates geometry management to the packer. *pack_options* may include:
expand

When true, *w* expands to fill any space not otherwise used in *w*'s parent.
fill

Determines whether *w* fills any extra space allocated to it by the packer, or keeps its own minimal dimensions: NONE (default), X (fill only horizontally), Y (fill only vertically), or BOTH (fill both horizontally and vertically).
side

Determines which side of the parent *w* packs against: TOP (default), BOTTOM, LEFT, or RIGHT. To avoid confusion, don't mix different values for option side= in widgets that are children of the same container. When more than one child requests the same side (for example TOP), the rule is first come, first served: the first child packs at the top, the second child packs second from the top, and so on.

pack_forget

```
w.pack_forget(  )
```

The packer forgets about *w*. *w* remains alive but invisible, and you may show *w* again later (by calling *w*.pack again, or perhaps *w*.grid or *w*.place).

pack_info

```
w.pack_info(  )
```

Returns a dictionary with the current *pack_options* of *w*.

16.8.2 The Gridder

16.9 Tkinter Events

So far, we've seen only the most elementary kind of event handling: the callbacks performed on callables installed with the command= option of buttons and menu entries of various kinds. Tkinter also lets you install callables to call back when needed to handle a variety of events. However, Tkinter does not let you create your own custom events; you are limited to working with events predefined by Tkinter itself.

16.9.1 The Event Object

General event callbacks must accept one argument *event* that is a Tkinter event object. Such an event object has several attributes describing the event:

char

A single-character string that is the key's code (only for keyboard events)

keysym

A string that is the key's symbolic name (only for keyboard events)

num

Button number (only for mouse-button events); 1 and up

x, y

Mouse position, in pixels, relative to the upper left corner of the widget

x_root , y_root

Mouse position, in pixels, relative to the upper left corner of the screen

widget

The widget in which the event has occurred

16.9.2 Binding Callbacks to Events

To bind a callback to an event in a widget *w*, call *w*.bind, describing the event with a string, usually enclosed in angle brackets ('<...>'). The following example prints 'Hello World' each time the user presses the Enter key:

```
from Tkinter import *

root = Tk(  )
def greet(*ignore): print 'Hello World'
root.bind('<Return>', greet)
root.mainloop(  )
```

Method tag_bind of classes Canvas and Text, covered earlier in this chapter, lets you bind event callbacks to specific sets of items of a Canvas instance, or to ranges within a Text instance.

16.9.3 Event Names

Frequently used event names, which are almost all enclosed in angle brackets, fall into a few categories.

16.9.3.1 Keyboard events

Key

Chapter 17. Testing, Debugging, and Optimizing

You're not finished with a programming task when you're done writing the code: you're finished when your code is running correctly and with acceptable performance. *Testing* means verifying that your code is running correctly by exercising the code under known conditions and checking that the results are as expected. *Debugging* means discovering the causes of incorrect behavior and removing them (the removal is often easy once you have figured out the causes).

Optimizing is often used as an umbrella term for activities meant to ensure acceptable performance. Optimizing breaks down into *benchmarking* (measuring performance for given tasks and checking that it's within acceptable bounds), *profiling* (instrumenting the program to find out what parts are performance bottlenecks), and optimizing proper (removing bottlenecks to make overall program performance acceptable). Clearly, you can't remove performance bottlenecks until you've found out where they are (using profiling), which in turn requires knowing that there are performance problems (using benchmarking).

All of these tasks are large and important, and each could fill a book by itself. This chapter does not explore every related technique and implication; it focuses on Python-specific techniques, approaches, and tools.

17.1 Testing

In this chapter, I distinguish between two rather different kinds of testing: unit testing and system testing. Testing is a rich and important field, and even more distinctions could be drawn, but my goal is to focus on the issues of most immediate importance to software developers.

17.1.1 Unit Testing and System Testing

Unit testing means writing and running tests to exercise a single module or an even smaller unit, such as a class or function. *System testing* (also known as functional testing) involves running an entire program with known inputs. Some classic books on testing draw the distinction between white-box testing, done with knowledge of a program's internals, and black-box testing, done from the outside. This classic viewpoint parallels the modern one of unit versus system testing.

Unit and system testing serve different goals. Unit testing proceeds apace with development; you can and should test each unit as you're developing it. Indeed, one modern approach is known as *test-first coding*: for each feature that your program must have, you first write unit tests, and only then do you proceed to write code that implements the feature. Test-first coding seems a strange approach, but it has several advantages. For example, it ensures that you won't omit unit tests for some feature. Further, test-first coding is helpful because it urges you to focus first on what tasks a certain function, class, or method should accomplish, and to deal only afterwards with implementing that function, class, or method. In order to test a unit, which may depend on other units not yet fully developed, you often have to write stubs, which are fake implementations of various units' interfaces that give known and correct responses in cases needed to test other units.

System testing comes afterwards, since it requires the system to exist with some subset of system functionality believed to be in working condition. System testing provides a sanity check: given that each module in the program works properly (passes unit tests), does the whole program work? If each unit is okay but the system as a whole is not, there is a problem with integration between units. For this reason, system testing is also known as integration testing.

System testing is similar to running the system in production use except that you fix the inputs in advance, so any problems you find are easy to reproduce. The cost of failure in system testing is lower than in production use, since outputs from system testing are not used to make decisions, control external systems, and so on. Rather, outputs from system testing are systematically compared with the outputs that the system should produce given the known inputs. The purpose of the whole procedure is to find discrepancies between what the program should do and what the program actually does in a cheap and reproducible way.

Failures discovered by system testing, just like system failures in production use, reveal defects in unit tests as well as defects in the code. Unit testing may have been insufficient; a module's unit tests may have failed to exercise all needed functionality of that module. In this case, the unit tests clearly need to be beefed up.

More often, failures in system testing reveal communication problems within the development team: a module may correctly implement a certain interface functionality, but another module expects different functionality. This kind of problem (an integration problem in the strict sense) is harder to pinpoint in unit testing. In good development practice, unit tests must run often, so it is crucial that they run fast. It's therefore essential that each unit can assume other units are working correctly and as expected.

17.2 Debugging

Since Python's development cycle is so fast, the most effective way to debug is often to edit your code to make it output relevant information at key points. Python has many ways to let your code explore its own state in order to extract information that may be relevant for debugging. The inspect and traceback modules specifically support such exploration, which is also known as reflection or introspection.

Once you have obtained debugging-relevant information, statement print is often the simplest way to display it. You can also log debugging information to files. Logging is particularly useful for programs that run unattended for a long time, as is typically the case for server programs. Displaying debugging information is like displaying other kinds of information, as covered in Chapter 10 and Chapter 16, and similarly for logging it, as covered in Chapter 10 and Chapter 11. Python 2.3 will also include a module specifically dedicated to logging. As covered in Chapter 8, rebinding attribute excepthook of module sys lets your program log detailed error information just before your program is terminated by a propagating exception.

Python also offers hooks enabling interactive debugging. Module pdb supplies a simple text-mode interactive debugger. Other interactive debuggers for Python are part of integrated development environments (IDEs), such as IDLE and various commercial offerings. However, I do not cover IDEs in this book.

17.2.1 The inspect Module

The inspect module supplies functions to extract information from all kinds of objects, including the Python call stack (which records all function calls currently executing) and source files. At the time of this writing, module inspect is not yet available for Jython. The most frequently used functions of module inspect are as follows.

getargspec, formatargspec

```
getargspec(f)
```

f is a function object. getargspec returns a tuple with four items (*arg_names*, *extra_args*, *extra_kwds*, *arg_defaults*). *arg_names* is the sequence of names of *f*'s formal arguments. *extra_args* is the name of the special formal argument of the form **args*, or None if *f* has no such special argument. *extra_kwds* is the name of the special formal argument of the form ***kwds*, or None if *f* has no such special argument. *arg_defaults* is the tuple of default values for *f*'s arguments. You can deduce other details about *f*'s signature from getargspec's results. For example, *f* has len(*arg_names*)-len(*arg_defaults*) mandatory arguments, and the names of *f*'s optional arguments are the strings that are the items of the list slice *arg_names*[-len(*arg_defaults*):].

formatargspec accepts one to four arguments that are the same as the items of the tuple that getargspec returns, and returns a formatted string that displays this information. Thus, formatargspec(*getargspec(*f*)) returns a formatted string with *f*'s formal arguments (i.e., *f*'s *signature*) in parentheses, as used in the def statement that created *f*.

getargvalues, formatargvalues

```
getargvalues(f)
```

f is a frame object, for example the result of a call to the function _getframe in module sys (covered in Chapter 8) or to function currentframe in module inspect. getargvalues returns a tuple with four items (*arg_names*, *extra_args*, *extra_kwds*, *locals*). *arg_names* is the sequence of names of *f*'s function's formal arguments. *extra_args* is the name of the special formal argument of form **args*, or None if *f*'s function has no such special argument. *extra_kwds*

17.3 The warnings Module

Warnings are messages about errors or anomalies that may not be serious enough to be worth disrupting the program's control flow (as would happen by raising a normal exception). The warnings module offers you fine-grained control over which warnings are output and what happens to them. Your code can conditionally output a warning by calling function warn in module warnings. Other functions in the module let you control how warnings are formatted, set their destinations, and conditionally suppress some warnings (or transform some warnings into exceptions).

17.3.1 Classes

Module warnings supplies several exception classes representing warnings. Class Warning subclasses Exception and is the base class for all warnings. You may define your own warning classes; they must subclass Warning, either directly or via one of its other existing subclasses, which are:

DeprecationWarning

Using deprecated features only supplied for backward compatibility

RuntimeWarning

Using features whose semantics are error-prone

SyntaxWarning

Using features whose syntax is error-prone

UserWarning

Other user-defined warnings that don't fit any of the above cases

17.3.2 Objects

In the current version of Python, there are no concrete warning objects. A warning is composed of a *message* (a text string), a *category* (a subclass of Warning), and two pieces of information that identify where the warning was raised from: *module* (name of the module raising the warning) and *lineno* (line number of the source code line raising the warning). Conceptually, you may think of these as attributes of a warning object *w*, and I use attribute notation later for clarity, but no specific warning object *w* actually exists.

17.3.3 Filters

At any time, module warnings keeps a list of active filters for warnings. When you import warnings for the first time in a run, the module examines sys.warnoptions to determine the initial set of filters. You can run Python with option -W to set sys.warnoptions for a given run. Do not rely on the initial set of filters being held specifically in sys.warnoptions, as this is an implementation aspect that may change in future releases of Python.

As each warning *w* occurs, warnings tests *w* against each filter until a filter matches. The matching filter determines what happens to *w*. Each filter is a tuple of five items. The first item, *action*, is a string that defines what happens on a match. The other four items, *message*, *category*, *module*, and *lineno*, control what it means for *w* to match the filter, and all conditions must be satisfied for a match. Here are the meanings of these items (using attribute notation to indicate conceptual attributes of *w*):

message

17.4 Optimization

"First make it work. Then make it right. Then make it fast." This quotation, often with slight variations, is widely known as the golden rule of programming. As far as I've been able to ascertain, the quotation is attributed to Kent Beck, who credits his father with it. Being widely known makes the principle no less important, particularly because it's more honored in the breach than in the observance. A negative form, slightly exaggerated for emphasis, is in a quotation by Don Knuth: "Premature optimization is the root of all evil in programming."

Optimization is premature if your code is not working yet. First make it work. Optimization is also premature if your code is working but you are not satisfied with the overall architecture and design. Remedy structural flaws before worrying about optimization: first make it work, then make it right. These first two steps are not optional—working, well-architected code is always a must.

In contrast, you don't always need to make it fast. Benchmarks may show that your code's performance is already acceptable after the first two steps. When performance is not acceptable, profiling often shows that all performance issues are in a small subset, perhaps 10% to 20% of the code where your program spends 80% or 90% of the time. Such performance-crucial regions of your code are also known as its *bottlenecks*, or *hot spots*. It's a waste of effort to optimize large portions of code that account for, say, 10% of your program's running time. Even if you made that part run 10 times as fast (a rare feat), your program's overall runtime would only decrease by 9%, a speedup no user will even notice. If optimization is needed, focus your efforts where they'll matter, on bottlenecks. You can optimize bottlenecks while keeping your code 100% pure Python. In some cases, you can resort to recoding some computational bottlenecks as Python extensions, potentially gaining even better performance.

17.4.1 Developing a Fast-Enough Python Application

Start by designing, coding, and testing your application in Python, often using some already available extension modules. This takes much less time than it would take with a classic compiled language. Then benchmark the application to find out if the resulting code is fast enough. Often it is, and you're done—congratulations!

Since much of Python itself is coded in highly optimized C, as are many of its standard and extension modules, your application may even turn out to be already faster than typical C code. However, if the application is too slow, you need to re-examine your algorithms and data structures. Check for bottlenecks due to application architecture, network traffic, database access, and operating system interactions. For typical applications, each of these factors is more likely than language choice to cause slowdowns. Tinkering with large-scale architectural aspects can often speed up an application dramatically, and Python is an excellent medium for such experimentation.

If your program is still too slow, you should profile it to find out where the time is going. Applications often exhibit computational bottlenecks—small areas of the source code, generally between 10% and 20%, which account for 80% or more of the running time. You can now optimize the bottlenecks, applying the techniques suggested in the rest of this chapter.

If normal Python-level optimizations still leave some outstanding computational bottlenecks, you can recode them as Python extension modules, as covered in Chapter 24. In the end, your application will run at roughly the same speed as if you had coded it all in C, C++, or Fortran—or faster, when large-scale experimentation has let you find a better architecture. Your overall programming productivity with this process is not much less than if you coded everything in Python. Future changes and maintenance are easy, since you use Python to express the overall structure of the program, and lower-level, harder-to-maintain languages only for a few specific computational bottlenecks.

Part IV: Network and Web Programming

Chapter 18. Client-Side Network Protocol Modules

A program can work on the Internet as a *client* (a program that accesses resources) or as a *server* (a program that makes services available). Both kinds of program deal with protocol issues, such as how to access and communicate data, and with data formatting issues. For order and clarity, the Python library deals with these issues in several different modules. This book will cover the topics in separate chapters. This chapter deals with the modules in the Python library that support protocol issues of client programs.

Nowadays, data access can often be achieved most simply through Uniform Resource Locators (URLs). Python supports URLs with modules urlparse, urllib, and urllib2. For rarer cases, when you need fine-grained control of data access protocols normally accessed via URLs, Python supplies modules httplib and ftplib. Protocols for which URLs are often insufficient include mail (modules poplib and smtplib), Network News (module nntplib), and Telnet (module telnetlib). Python also supports the XML-RPC protocol for distributed computing with module xmlrpclib.

18.1 URL Access

A URL identifies a resource on the Internet. A URL is a string composed of several optional parts, called components, known as scheme, location, path, query, and fragment. A URL with all its parts looks something like:

```
scheme://lo.ca.ti.on/pa/th?query#fragment
```

For example, in http://www.python.org:80/faq.cgi?src=fie, the scheme is *http*, the location is *www.python.org:80*, the path is */faq.cgi*, the query is *src=fie*, and there is no fragment. Some of the punctuation characters form a part of one of the components they separate, while others are just separators and are part of no component. Omitting punctuation implies missing components. For example, in mailto:me@you.com, the scheme is *mailto*, the path is *me@you.com*, and there is no location, query, or fragment. The missing // means the URL has no location part, the missing ? means it has no query part, and the missing # means it has no fragment part.

18.1.1 The urlparse Module

The urlparse module supplies functions to analyze and synthesize URL strings. In Python 2.2, the most frequently used functions of module urlparse are urljoin, urlsplit, and urlunsplit.

urljoin

urljoin(*base_url_string*, *relative_url_string*)

Returns a URL string *u*, obtained by joining *relative_url_string*, which may be relative, with *base_url_string*. The joining procedure that urljoin performs to obtain its result *u* may be summarized as follows:

- When either of the argument strings is empty, *u* is the other argument.
- When *relative_url_string* explicitly specifies a scheme different from that of *base_url_string*, *u* is *relative_url_string*. Otherwise, *u*'s scheme is that of *base_url_string*.
- When the scheme does not allow relative URLs (e.g., *mailto*), or *relative_url_string* explicitly specifies a location (even when it is the same as the location of *base_url_string*), all other components of *u* are those of *relative_url_string*. Otherwise, *u*'s location is that of *base_url_string*.
- *u*'s path is obtained by joining the paths of *base_url_string* and *relative_url_string* according to standard syntax for absolute and relative URL paths. For example:

```
import urlparse
urlparse.urljoin(
    'http://somehost.com/some/path/here',
    '../other/path')
# Result is: 'http://somehost.com/some/other/path'
```

urlsplit

18.2 Email Protocols

Most email today is sent via servers that implement the Simple Mail Transport Protocol (SMTP) and received via servers that implement the Post Office Protocol Version 3 (POP3). These protocols are supported by the Python standard library modules smtplib and poplib, respectively. Some servers, instead of or in addition to POP3, implement the richer and more advanced Internet Message Access Protocol Version 4 (IMAP4), supported by the Python standard library module imaplib, which I do not cover in this book.

18.2.1 The poplib Module

The poplib module supplies a class POP3 to access a POP mailbox.

POP3

```
class POP3(host,port=110)
```

Returns an instance *p* of class POP3 connected to the given *host* and *port*.

Instance *p* supplies many methods, of which the most frequently used are the following.

dele

```
p.dele(msgnum)
```

Marks message *msgnum* for deletion. The server performs deletions when this connection terminates by a call to method *quit*. Returns the response string.

list

```
p.list(msgnum=None)
```

Returns a pair (*response*,*messages*) where *response* is the response string and *messages* is a list of strings, each of two words '*msgnum bytes*', giving the message number and the length in bytes of each message in the mailbox. When *msgnum* is not None, list *messages* has only one item, a string with two words: *msgnum* as requested, and the length *bytes*.

pass_

```
p.pass_(password)
```

Sends the password. Must be called after method user. The trailing underscore in the function's name is necessary because pass is a Python keyword. Returns the response string.

quit

```
p.quit(  )
```

Ends the session and performs the deletions that were requested by calls to method dele. Returns the response string.

18.3 The HTTP and FTP Protocols

Modules urllib and urllib2 are most often the handiest ways to access servers for *http*, *https*, and *ftp* protocols. The Python standard library also supplies specific modules to use for these data access protocols.

18.3.1 The httplib Module

Module httplib supplies a class HTTPConnection to connect to an HTTP server.

HTTPConnection

```
class HTTPConnection(host,port
=80)
```

Returns an instance *h* of class HTTPConnection, ready for connection (but not yet connected) to the given *host* and *port*.

Instance *h* supplies several methods, of which the most frequently used are the following.

close

```
h.close(  )
```

Closes the connection to the HTTP server.

getresponse

```
h.getresponse(  )
```

Returns an instance *r* of class HTTPResponse, which represents the response received from the HTTP server. Call after method request has returned. Instance *r* supplies the following attributes and methods:
r.getheader(*name*,*default*=None)

Returns the contents of header *name*, or *default* if no such header exists.
r.msg

An instance of class Message of module mimetools, covered in Chapter 21. You can use *r*.msg to access the response's headers and body.
r.read( )

Returns a string that is the body of the server's response.
r.reason

The string that the server gave as the reason for errors or anomalies. If the request was successful, *r*.reason could, for example, be 'OK'.
r.status

An integer, the status code that the server returned. If the request was successful, *r*.status should be between 200 and 299 according to the HTTP standards. Values between 400 and 599 are typical error codes, again according to

18.4 Network News

Network News, also known as Usenet News, is mostly transmitted with the Network News Transport Protocol (NNTP). The Python standard library supports this protocol in its module nntplib. The nntplib module supplies a class NNTP to connect to an NNTP server.

NNTP

```
class NNTP(
    host,port=119,user=None,
passwd=None,readermode=False)
```

Returns an instance *n* of class NNTP connected to the given *host* and *port*, and optionally authenticated with the given *user* and *passwd* if *user* is not None. When *readermode* is True, also sends a 'mode reader' command; you may need this, depending on what NNTP server you connect to and on what NNTP commands you send to that server.

18.4.1 Response Strings

An instance *n* of NNTP supplies many methods. Each of *n*'s methods returns a tuple whose first item is a string (referred to as *response* in the following section) that is the response from the NNTP server to the NNTP command corresponding to the method (method post just returns the *response* string, not a tuple). Each method returns the *response* string just as the NNTP server supplies it. The string starts with an integer in decimal form (the integer is known as the return code), followed by a space, followed by more text.

For some commands, the extra text after the return code is just a comment or explanation supplied by the NNTP server. For other commands, the NNTP standard specifies the format of the text that follows the return code on the response line. In those cases, the relevant method also parses the text in question, yielding other items in the method's resulting tuple, so your code need not perform such parsing itself; rather, you can just access further items in the method's result tuple, as specified in the following sections.

Return codes of the form 2*xx*, for any two digits *xx*, are success codes (i.e., they indicate that the corresponding NNTP command succeeded). Return codes of other forms, such as 4*xx* and 5*xx*, indicate failures in the corresponding NNTP command. In these cases, the method does not return a result. Rather, the method raises an instance of exception class nntplib.NNTPError or some appropriate subclass of it, such as NNTPTemporaryError for errors that may (or may not) be automatically resolved if you try the operation again, or NNTPPermanentError for errors that are sure to occur again if you retry. When a method of an NNTP instance raises an NNTPError instance *e*, the server's response string, starting with a return code such as 4*xx*, is accessible as str(*e*).

18.4.2 Methods

The most frequently used methods of an NNTP instance *n* are as follows.

article

```
n.article(id)
```

id is a string, either an article ID enclosed in angle brackets (<>) or an article number in the current group. Returns a tuple of three strings and a list (*response*,*number*,*id*,*list*), where *number* is the article number in the current group, *id*

18.5 Telnet

Telnet is an old protocol, specified by RFC 854 (see http://www.faqs.org/rfcs/rfc854.html), and normally used for interactive user sessions. The Python standard library supports this protocol in its module telnetlib. Module telnetlib supplies a class Telnet to connect to a Telnet server.

Telnet

```
class Telnet(host=None,port=23)
```

Returns an instance *t* of class Telnet. When *host* (and optionally *port*) is given, implicitly calls *t*.open(*host*,*port*).

Instance *t* supplies many methods, of which the most frequently used are as follows.

close

```
t.close(  )
```

Closes the connection.

expect

```
t.expect(res,timeout=None)
```

Reads data from the connection until it matches any of the regular expressions that are the items of list *res*, or until *timeout* seconds elapse when *timeout* is not None. Regular expressions and match objects are covered in Chapter 9 . Returns a tuple of three items (*i*,*mo*,*txt*), where *i* is the index in *res* of the regular expression that matched, *mo* is the match object, and *txt* is all the text read until the match, included. Raises EOFError when the connection is closed and no data is available; otherwise, when it gets no match, returns (-1,None,*txt*), where *txt* is all the text read, or possibly '' if nothing was read before a timeout. Results are non-deterministic if more than one item in *res* can match, or if any of the items in *res* include greedy parts (such as '.*').

interact

```
t.interact(  )
```

Enters interactive mode, connecting standard input and output to the two channels of the connection, like a dumb Telnet client.

open

```
t.open(host,port=23)
```

Connects to a Telnet server on the given *host* and *port*. Call once per instance *t*, as *t*'s first method call. Don't call if *host* was given on creation.

read_all

18.6 Distributed Computing

There are many standards for distributed computing, from simple Remote Procedure Call (RPC) ones to rich object-oriented ones such as CORBA. You can find several third-party Python modules supporting these standards on the Internet.

The Python standard library comes with support for both server and client use of a simple yet powerful standard known as XML-RPC. For in-depth coverage of XML-RPC, I recommend the book Programming Web Services with XML-RPC, by Simon St. Laurent and Joe Johnson (O'Reilly). XML-RPC uses HTTP as the underlying transport and encodes requests and replies in XML. For server-side support, see Section 19.2.2.4 in Chapter 19. Client-side support is supplied by module xmlrpclib.

The xmlrcplib module supports a class ServerProxy, which you instantiate to connect to an XML-RPC server. An instance *s* of ServerProxy is a proxy for the server it connects to. In other words, you call arbitrary methods on *s*, and *s* packages up the method name and argument values as an XML-RPC request, sends the request to the XML-RPC server, receives the server's response, and unpackages the response as the method's result. The arguments to such method calls can be of any type supported by XML-RPC:

Boolean

Constant attributes True and False of module xmlrpclib (since module xlmrpclib predates the introduction of bool into Python, it does not use Python's built-in True and False values for this purpose)

Integers, floating-point numbers, strings, arrays

Passed and returned as Python int, float, Unicode, and list values

Structures

Passed and returned as Python dict values whose keys must be strings

Dates

Passed as instances of class xmlrpclib.DateTime; value is represented in seconds since the epoch, as in module time (see Chapter 12)

Binary data

Passed as instances of class xmlrpclib.Binary; value is an arbitrary string of bytes

Module xmlrpclib supplies two factory functions.

binary

`binary(`*`bytestring`*`)`

Creates and returns an instance of Binary wrapping the given *bytestring*.

boolean

`boolean(`*`x`*`)`

Creates and returns an instance of Boolean with the truth value of *x*.

Chapter 19. Sockets and Server-Side Network Protocol Modules

To communicate with the Internet, programs use devices known as *sockets*. The Python library supports sockets through module socket, as well as wrapping them into higher-level modules covered in Chapter 18. To help you write server programs, the Python library also supplies higher-level modules to use as frameworks for socket servers. Standard and third-party Python modules and extensions also support timed and asynchronous socket operations. This chapter covers socket, the server-side framework modules, and the essentials of other, more advanced modules.

The modules covered in this chapter offer many conveniences compared to C-level socket programming. However, in the end, the modules rely on native socket functionality supplied by the underlying operating system. While it is often possible to write effective network clients by using just the modules covered in Chapter 18, without needing to understand sockets, writing effective network servers most often does require some understanding of sockets. Thus, the lower-level module socket is covered in this chapter and not in Chapter 18, even though both clients and servers use sockets.

However, I only cover the ways in which module socket lets your program access sockets; I do not try to impart the detailed understanding of sockets, and of other aspects of network behavior independent of Python, that you may need to make use of socket's functionality. To understand socket behavior in detail on any kind of platform, I recommend W. Richard Stevens' Unix Network Programming, Volume 1 (Prentice-Hall). Higher-level modules are simpler and more powerful, but a detailed understanding of the underlying technology is always useful, and sometimes it can prove indispensable.

19.1 The socket Module

The socket module supplies a factory function, also named socket, that you call to generate a socket object *s*. You perform network operations by calling methods on *s*. In a client program, you connect to a server by calling *s* .connect. In a server program, you wait for clients to connect by calling *s*.bind and *s*.listen. When a client requests a connection, you accept the request by calling *s*.accept, which returns another socket object *s1* connected to the client. Once you have a connected socket object, you transmit data by calling its method send, and receive data by calling its method recv.

Python supports both current Internet Protocol (IP) standards. IPv4 is more widespread, while IPv6 is newer. In IPv4, a network address is a pair (*host*,*port*), where *host* is a Domain Name System (DNS) hostname such as 'www.python.org' or a dotted-quad IP address string such as '194.109.137.226'. *port* is an integer indicating a socket's port number. In IPv6, a network address is a tuple (*host*, *port*, *flowinfo*, *scopeid*). Since IPv6 infrastructure is not yet widely deployed, I do not cover IPv6 further in this book. When *host* is a DNS hostname, Python implicitly looks up the name, using your platform's DNS infrastructure, and uses the dotted-quad IP address corresponding to that name.

Module socket supplies an exception class error. Functions and methods of the module raise error instances to diagnose socket-specific errors. Module socket also supplies many functions. Several of these functions translate data, such as integers, between your host's native format and network standard format. The higher-level protocol that your program and its counterpart are using on a socket determines what kind of conversions you must perform.

19.1.1 socket Functions

The most frequently used functions of module socket are as follows.

getfqdn

```
getfqdn(host='')
```

Returns the fully qualified domain name string for the given *host*. When *host* is '', returns the fully qualified domain name string for the local host.

gethostbyaddr

```
gethostbyaddr(ipaddr)
```

Returns a tuple with three items (*hostname*, *alias_list*, *ipaddr_list*). *hostname* is a string, the primary name of the host whose IP dotted-quad address you pass as string *ipaddr*. *alias_list* is a list of 0 or more alias names for the host. *ipaddr_list* is a list of one or more dotted-quad addresses for the host.

gethostbyname_ex

```
gethostbyname_ex(hostname)
```

Returns the same results as gethostbyaddr, but takes as an argument a *hostname* string that can be either an IP dotted-quad address or a DNS name.

htonl

19.2 The SocketServer Module

The Python library supplies a framework module, SocketServer, to help you implement Internet servers. SocketServer supplies server classes TCPServer, for connection-oriented servers using TCP, and UDPServer, for datagram-oriented servers using UDP, with the same interface.

An instance *s* of either TCPServer or UDPServer supplies many attributes and methods, and you can subclass either class and override some methods to architect your own specialized server framework. However, I do not cover such advanced and rarely used possibilities in this book.

Classes TCPServer and UDPServer implement synchronous servers, able to serve one request at a time. Classes ThreadingTCPServer and ThreadingUDPServer implement threaded servers, spawning a new thread per request. You are responsible for synchronizing the resulting threads as needed. Threading is covered in Chapter 14.

19.2.1 The BaseRequestHandler Class

For normal use of SocketServer, subclass the BaseRequestHandler class provided by SocketServer and override the handle method. Then, instantiate a server class, passing the address pair on which to serve and your subclass of BaseRequestHandler. Finally, call method serve_forever on the server class instance.

An instance *h* of BaseRequestHandler supplies the following methods and attributes.

client_address

The *h*.client_address attribute is the pair (*host*,*port*) of the client, set by the base class at connection.

handle

```
h.handle(  )
```

Your subclass overrides this method, called by the server, on a new instance of your subclass for each new incoming request. Typically, for a TCP server, your implementation of handle conducts a conversation with the client on socket *h*.request to service the request. For a UDP server, your implementation of handle examines the datagram in *h* .request[0] and sends a reply string with *h*.request[1].sendto.

request

For a TCP server, the *h*.request attribute is the socket connected to the client. For a UDP server, the *h*.request attribute is a pair (*data*,*sock*), where *data* is the string of data the client sent as a request (up to 8192 bytes) and *sock* is the server socket. Your handle method typically calls method sendto on *sock* to send a reply to the client.

server

The *h*.server attribute is the instance of the server class that instantiated this handler object.

19.3 Event-Driven Socket Programs

Socket programs, particularly servers, must often be ready to perform many tasks at once. Example 19-1 accepts a connection request, then serves a single client until that client has finished—other connection requests must wait. This is not acceptable for servers in production use. Clients cannot wait too long: the server must be able to service multiple clients at once.

One approach that lets your program perform several tasks at once is threading, covered in Chapter 14. Module SocketServer optionally supports threading, as covered earlier in this chapter. An alternative to threading that can offer better performance and scalability is *event-driven* (also known as *asynchronous*) programming.

An event-driven program sits in an event loop, where it waits for events. In networking, typical events are "a client requests connection," "data arrived on a socket," and "a socket is available for writing." The program responds to each event by executing a small slice of work to service that event, then goes back to the event loop to wait for the next event. The Python library supports event-driven network programming with low-level select module and higher-level asyncore and asynchat modules. Even more complete support for event-driven programming is in the Twisted package (available at http://www.twistedmatrix.com), particularly in subpackage twisted.internet.

19.3.1 The select Module

The select module exposes a cross-platform low-level function that lets you implement high-performance asynchronous network servers and clients. Module select offers additional platform-dependent functionality on Unix-like platforms, but I cover only cross-platform functionality in this book.

select

```
select(inputs,outputs,excepts,
timeout=None)
```

inputs, *outputs*, and *excepts* are lists of socket objects waiting for input events, output events, and exceptional conditions, respectively. *timeout* is a float, the maximum time to wait in seconds. When *timeout* is None, there is no maximum wait: select waits until one or more objects receive events. When *timeout* is 0, select returns at once, without waiting.

select returns a tuple with three items (*i*,*o*,*e*). *i* is a list of zero or more of the items of *inputs*, those that received input events. *o* is a list of zero or more of the items of *outputs*, those that received output events. *e* is a list of zero or more of the items of *excepts*, those that received exceptional conditions (i.e., out-of-band data). Any or all of *i*, *o*, and *e* can be empty, but at least one of them is non-empty if *timeout* is None.

In addition to sockets, you can have in lists *inputs*, *outputs*, and *excepts* other objects that supply a method fileno, callable without arguments, returning a socket's file descriptor. For example, the server classes of module SocketServer, covered earlier in this chapter, follow this protocol. Therefore, you can have instances of those classes in the lists. On Unix-like platforms, select.select has wider applicability, since it can also accept file descriptors that do not refer to sockets. On Windows, however, select.select can accept only file descriptors that do refer to sockets.

Example 19-6 uses module select to reimplement the server of Example 19-1 with the added ability to serve any number of clients simultaneously.

Chapter 20. CGI Scripting and Alternatives

When a web browser (or other web client) requests a page from a web server, the server may return either static or dynamic content. Serving dynamic content involves server-side web programs that generate and deliver content on the fly, often based on information that is stored in a database. The one longstanding Web-wide standard for server-side programming is known as CGI, which stands for Common Gateway Interface. In server-side programming, a client sends a structured request to a web server. The server runs another program, passing the content of the request. The server captures the output of the other program, and sends that output to the client as the response to the original request. In other words, the server's role is that of a gateway between the client and the other program. The other program is called a CGI program or CGI script.

CGI enjoys the typical advantages of standards. When you program to the CGI standard, your program can be deployed on different web servers, and work despite the differences. This chapter focuses on CGI scripting in Python. It also mentions the downsides of CGI (basically, issues of scalability under high load) and some of the alternative, nonstandard server-side architectures that you can use instead of CGI.

This chapter assumes that you are familiar with both HTML and HTTP. For reference material on both of these standards, see Webmaster in a Nutshell, by Stephen Spainhour and Robert Eckstein (O'Reilly). For detailed coverage of HTML, I recommend HTML & XHTML: The Definitive Guide, by Chuck Musciano and Bill Kennedy (O'Reilly). And for additional coverage of HTTP, see the HTTP Pocket Reference, by Clinton Wong (O'Reilly).

20.1 CGI in Python

CGI's standardization lets you use any language to code CGI scripts. Python is a very-high-level, high-productivity language, and thus quite suitable for CGI coding. The Python standard library supplies modules to handle typical CGI-related tasks.

20.1.1 Form Submission Methods

CGI scripts are often used to handle HTML form submissions. In this case, the action attribute of the form tag specifies a URL for a CGI script to handle the form, and the method attribute is either GET or POST, indicating how the form data is sent to the script. According to the CGI standard, the GET method should be used for forms without side effects, such as asking the server to query a database and display the results, while the POST method is meant for forms with side effects, such as asking the server to update a database. In practice, however, GET is also often used to create side effects. The distinction between GET and POST in practical use is that GET encodes the form's contents as a query string joined to the action URL to form a longer URL, while POST transmits the form's contents as an encoded stream of data, which a CGI script sees as the script's standard input.

The GET method is slightly faster. You can use a fixed GET-form URL wherever you can use a hyperlink. However, GET cannot send large amounts of data to the server, since many clients and servers limit URL lengths (you're safe up to about 200 bytes). The POST method has no size limits. You must use POST when the form contains input tags with type=file—the form tag must then have enctype=multipart/form-data.

The CGI standard does not specify whether a single script can access both the query string (used for GET) and the script's standard input (used for POST). Many clients and servers let you get away with it, but relying on this nonstandard practice may negate the portability advantages that you would otherwise get from the fact that CGI is a standard. Python's standard module cgi, covered in the next section, recovers form data from the query string only, when any query string is present; otherwise, when no query string is present, cgi recovers form data from standard input.

20.1.2 The cgi Module

The cgi module supplies several functions and classes, mostly for backward compatibility or unusual needs. CGI scripts use one function and one class from module cgi.

escape

```
escape(str,quote=0)
```

Returns a copy of string *str*, replacing each occurrence of characters &, <, and > with the appropriate HTML entity (&amp;, &lt;, &gt;). When *quote* is true, escape also replaces double quote characters (") with &quot;. Function escape lets a script prepare arbitrary text strings for output within an HTML document, whether or not the strings contain characters that HTML interprets in special ways.

FieldStorage

```
class FieldStorage(
keep_blank_values=0)
```

20.2 Cookies

HTTP is a stateless protocol, meaning that it retains no session state between transactions. Cookies, as specified by the HTTP 1.1 standard, let web clients and servers cooperate to build a stateful session from a sequence of HTTP transactions.

Each time a server sends a response to a client's request, the server may initiate or continue a session by sending one or more Set-Cookie headers, whose contents are small data items called *cookies*. When a client sends another request to the server, the client may continue a session by sending Cookie headers with cookies previously received from that server or other servers in the same domain. Each cookie is a pair of strings, the name and value of the cookie, plus optional attributes. Attribute max-age is the maximum number of seconds the cookie should be kept. The client should discard saved cookies after their maximum age. If max-age is missing, then the client should discard the cookie when the user's interactive session ends.

Cookies have no intrinsic privacy nor authentication. Cookies travel in the clear on the Internet, and therefore are vulnerable to sniffing. A malicious client might return cookies different from cookies previously received. To use cookies for authentication or identification or to hold sensitive information, the server must encrypt and encode cookies sent to clients, and decode, decrypt, and verify cookies received back from clients.

Encryption, encoding, decoding, decryption, and verification may all be slow when applied to large amounts of data. Decryption and verification require the server to keep some amount of server-side state. Sending substantial amounts of data back and forth on the network is also slow. The server should therefore persist most state data locally, in files or databases. In most cases, a server should use cookies only as small, encrypted, verifiable keys confirming the identity of a user or session, using DBM files or a relational database (covered in Chapter 11) for session state. HTTP sets a limit of 2 KB on cookie size, but I suggest you normally use substantially smaller cookies.

20.2.1 The Cookie Module

The Cookie module supplies several classes, mostly for backward compatibility. CGI scripts normally use the following classes from module Cookie.

Morsel

A script does not directly instantiate class Morsel. However, instances of cookie classes hold instances of Morsel. An instance *m* of class Morsel represents a single cookie element: a key string, a value string, and optional attributes. *m* is a mapping. The only valid keys in *m* are cookie attribute names: 'comment', 'domain', 'expires', 'max-age', 'path', 'secure', and 'version'. Keys into *m* are case-insensitive. Values in *m* are strings, each holding the value of the corresponding cookie attribute.

SimpleCookie

```
class SimpleCookie(input=None)
```

A SimpleCookie instance *c* is a mapping. *c*'s keys are strings. *c*'s values are Morsel instances that wrap strings. *c*[*k*]= *v* implicitly expands to:

```
c[k]=Morsel(  ); c[k].set(k,str(v),str(v))
```

20.3 Other Server-Side Approaches

A CGI script runs as a new process each time a client requests it. Process startup time, interpreter initialization, connection to databases, and script initialization all add up to measurable overhead. On fast, modern server platforms, the overhead is bearable for light to moderate loads. On a busy server, CGI may not scale up well. Web servers support server-specific ways to reduce overhead, running scripts in processes that can serve for several hits rather than starting up a new CGI process per hit.

Microsoft's ASP (Active Server Pages) is a server extension leveraging a lower-level library, ISAPI, and Microsoft's COM technology. Most ASP pages are coded in the VBScript language, but ASP is language-independent. As the reptilian connection suggests, Python and ASP go very well together, as long as Python is installed with the platform-specific win32all extensions, specifically ActiveScripting. Many other server extensions are cross-platform, not tied to specific operating systems.

The popular content server framework Zope (http://www.zope.org) is a Python application. If you need advanced content management features, Zope should definitely be among the solutions you consider. However, Zope is a large, rich, powerful system, needing a full book of its own to do it justice. Therefore, I do not cover Zope further in this book.

20.3.1 FastCGI

FastCGI lets you write scripts similar to CGI scripts, yet use each process to handle multiple hits, either sequentially or simultaneously in separate threads. FastCGI is available for Apache and other free web servers, but at the time of this writing not for Microsoft IIS. See http://www.fastcgi.com for FastCGI overviews and details. Go to http://alldunn.com/python/fcgi.py for a pure Python interface to FastCGI, letting scripts exploit FastCGI if available and fall back to normal CGI otherwise.

20.3.2 LRWP

Long-Running Web Processes (LRWP) are currently available only for Xitami (see http://www.xitami.org). Go to http://alldunn.com/python/lrwp.py for a pure Python module (by Robin Dunn, the architect of LRWP) that lets scripts exploit LRWP if available and fall back to normal CGI otherwise. LRWP peer processes connect to the web server via sockets. The server can use any number of peers that offer the same service. The server uses simple round-robin scheduling among equivalent available peers. If a request arrives when all peers are busy, the web server queues the request until a peer is free. This simple, clean protocol makes it easy to load-balance service requests among any number of hosts connected to the server's host by a fast, trusted local area network. Robin Dunn's article about LRWP, at http://www.imatix.com/html/xitami/index12.htm, gives architectural details and C and Python examples of LRWP peers.

20.3.3 PyApache and mod_python

Apache's architecture is modular. Besides CGI and FastCGI, other modules support Python server-side scripting with Apache. Simple, lightweight PyApache (http://bel-epa.com/pyapache/) focuses on letting you use CGI-like scripts with low overhead. mod_python (http://www.modpython.org) affords fuller access to Apache internals, including the ability to write authentication scripts. Both modules support the classic, widespread Apache 1.3 and the newer Apache 2.0.

20.3.4 Webware

Chapter 21. MIME and Network Encodings

What travels on a network are streams of bytes or text. However, what you want to send over the network often has more structure. The Multipurpose Internet Mail Extensions (MIME) and other encoding standards bridge the gap by specifying how to represent structured data as bytes or text. Python supports such encodings through many library modules, such as base64, quopri, uu, and the modules of the email package. This chapter covers these modules.

21.1 Encoding Binary Data as Text

Several kinds of media (e.g., email messages) contain only text. When you want to transmit binary data via such media, you need to encode the data as text strings. The Python standard library supplies modules that support the standard encodings known as Base 64, Quoted Printable, and UU.

21.1.1 The base64 Module

The base64 module supports the encoding specified in RFC 1521 as Base 64. The Base 64 encoding is a compact way to represent arbitrary binary data as text, without any attempt to produce human-readable results. Module base64 supplies four functions.

decode

```
decode(infile,outfile)
```

Reads text-file-like object *infile*, by calling *infile*.readline until end of file (i.e, until a call to *infile*.readline returns an empty string), decodes the Base 64-encoded text thus read, and writes the decoded data to binary-file-like object *outfile*.

decodestring

```
decodestring(s)
```

Decodes text string *s*, which contains one or more complete lines of Base 64-encoded text, and returns the byte string with the corresponding decoded data.

encode

```
encode(infile,outfile)
```

Reads binary-file-like object *infile*, by calling *infile*.read (for a few bytes at a time—the amount of data that Base 64 encodes into a single output line) until end of file (i.e, until a call to *infile*.read returns an empty string). Then it encodes the data thus read in Base 64, and writes the encoded text as lines to text-file-like object *outfile*. encode appends \n to each line of text it emits, including the last one.

encodestring

```
encodestring(s)
```

Encodes binary string *s*, which contains arbitrary bytes, and returns a text string with one or more complete lines of Base 64-encoded data. encodestring always returns a text string ending with \n.

21.1.2 The quopri Module

The quopri module supports the encoding specified in RFC 1521 as Quoted Printable (QP). QP can represent any binary data as text, but it's mainly intended for data that is textual, with a relatively modest amount of characters with

21.2 MIME and Email Format Handling

Python supplies the email package to handle parsing, generation, and manipulation of MIME files such as email messages, network news posts, and so on. The Python standard library also contains other modules that handle some parts of these jobs. However, the new email package offers a more complete and systematic approach to these important tasks. I therefore suggest you use package email, not the older modules that partially overlap with parts of email's functionality. Package email has nothing to do with receiving or sending email; for such tasks, see modules poplib and smtplib, covered in Chapter 18. Instead, package email deals with how you handle messages after you receive them or before you send them.

21.2.1 Functions in Package email

Package email supplies two factory functions returning an instance *m* of class email.Message.Message. These functions rely on class email.Parser.Parser, but the factory functions are handier and simpler. Therefore, I do not cover module Parser further in this book.

message_from_string

```
message_from_string(s)
```

Builds *m* by parsing string *s*.

message_from_file

```
message_from_file(f)
```

Builds *m* by parsing the contents of file-like object *f*, which must be open for reading.

21.2.2 The email.Message Module

The email.Message module supplies class Message. All parts of package email produce, modify, or use instances of class Message. An instance *m* of Message models a MIME message, including headers and a payload (data content). You can create *m*, initially empty, by calling class Message, which accepts no arguments. More often, you create *m* by parsing via functions message_from_string and message_from_file of module email, or by other indirect means such as the classes covered in "Creating Messages" later in this chapter. *m*'s payload can be a string, a single other instance of Message, or a list of other Message instances for a multipart message.

You can set arbitrary headers on email messages you're building. Several Internet RFCs specify headers that you can use for a wide variety of purposes. The main applicable RFC is RFC 2822 (see http://www.faqs.org/rfcs/rfc2822.html). An instance *m* of class Message holds headers as well as a payload. *m* is a mapping, with header names as keys and header value strings as values. The semantics of *m* as a mapping are rather different from those of a dictionary, to make *m* more convenient. *m*'s keys are case-insensitive. *m* keeps headers in the order in which you add them, and methods keys, values, and items return headers in that order. *m* can have more than one header named *key*—*m*[*key*] returns an arbitrary one of them, del *m*[*key*] deletes all of them. len(*m*) returns the total number of headers, counting duplicates, not just the number of distinct header names. If there is no header named *key*, *m*[*key*] returns None and does not raise KeyError (i.e., behaves like *m*.get(*key*)), and del *m*[*key*] is a no-operation.

An instance *m* of Message supplies the following attributes and methods dealing with *m*'s headers and payload.

Chapter 22. Structured Text: HTML

Most documents on the Web use HTML, the HyperText Markup Language. Markup is the insertion of special tokens, known as *tags*, in a text document to give structure to the text. HTML is an application of the large, general standard known as SGML, the Standard General Markup Language. In practice, many of the Web's documents use HTML in sloppy or incorrect ways. Browsers have evolved many practical heuristics over the years to try and compensate for this, but even so, it still often happens that a browser displays an incorrect web page in some weird way.

Moreover, HTML was never suitable for much more than presenting documents on a screen. Complete and precise extraction of the information in the document, working backward from the document's presentation, is often unfeasible. To tighten things up again, HTML has evolved into a more rigorous standard called XHTML. XHTML is very similar to traditional HTML, but it is defined in terms of XML and more precisely than HTML. You can handle XHTML with the tools covered in Chapter 23.

Despite the difficulties, it's often possible to extract at least some useful information from HTML documents. Python supplies the sgmllib, htmllib, and HTMLParser modules for the task of parsing HTML documents, whether this parsing is for the purpose of presenting the documents, or, more typically, as part of an attempt to extract information from them. Generating HTML and embedding Python in HTML are also frequent tasks. No standard Python library module supports HTML generation or embedding directly, but you can use normal Python string manipulation, and third-party modules can also help.

22.1 The sgmllib Module

The name of the sgmllib module is misleading: sgmllib parses only a tiny subset of SGML, but it is still a good way to get information from HTML files. sgmllib supplies one class, SGMLParser, which you subclass to override and add methods. The most frequently used methods of an instance *s* of your subclass *X* of SGMLParser are as follows.

close

`s.close(  )`

Tells the parser that there is no more input data. When *X* overrides close, *x*.close must call SGMLParser.close to ensure that buffered data get processed.

do_tag

`s.do_tag(attributes)`

X supplies a method with such a name for each *tag*, with no corresponding end tag, that *X* wants to process. *tag* must be in lowercase in the method name, but can be in any mix of cases in the parsed text. SGMLParser's handle_tag method calls do_*tag* as appropriate. *attributes* is a list of pairs (*name*,*value*), where *name* is each attribute's name, lowercased, and *value* is the value, processed to resolve entity references and character references and to remove surrounding quotes.

end_tag

`s.end_tag(  )`

X supplies a method with such a name for each *tag* whose end tag *X* wants to process. *tag* must be in lowercase in the method name, but can be in any mix of cases in the parsed text. *X* must also supply a method named start_*tag*, otherwise end_*tag* is ignored. SGMLParser's handle_endtag method calls end_*tag* as appropriate.

feed

`s.feed(data)`

Passes to the parser some of the text being parsed. The parser may process some prefix of the text, holding the rest in a buffer until the next call to *s*.feed or *s*.close.

handle_charref

`s.handle_charref(ref)`

Called to process a character reference '&#*ref*;'. SGMLParser's implementation of handle_charref handles decimal numbers in range(0,256), like:

```
def handle_charref(self, ref):
    try:
        c = chr(int(ref))
    except (TypeError, ValueError):
        self.unknown_charref(ref)
    else: self.handle_data(c)
```

22.2 The htmllib Module

The htmllib module supplies a class named HTMLParser that subclasses SGMLParser and defines start_*tag*, do_*tag*, and end_*tag* methods for tags defined in HTML 2.0. HTMLParser implements and overrides methods in terms of calls to methods of a formatter object, covered later in this chapter. You can subclass HTMLParser to add or override methods. In addition to the start_*tag*, do_*tag*, and end_*tag* methods, an instance *h* of HTMLParser supplies the following attributes and methods.

anchor_bgn

```
h.anchor_bgn(href,name,type)
```

Called for each <a> tag. *href*, *name*, and *type* are the string values of the tag's attributes with the same names. HTMLParser's implementation of anchor_bgn maintains a list of outgoing hyperlinks (i.e., *href* arguments of method *s* .anchor_bgn) in an instance attribute named *s*.anchorlist.

anchor_end

```
h.anchor_end(  )
```

Called for each </a> end tag. HTMLParser's implementation of anchor_end emits to the formatter a footnote reference that is an index within *s*.anchorlist. In other words, by default, HTMLParser asks the formatter to format an <a>/</a> tag pair as the text inside the tag, followed by a footnote reference number that points to the URL in the <a> tag. Of course, it's up to the formatter to deal with this formatting request.

anchorlist

The *h*.anchor_list attribute contains the list of outgoing hyperlink URLs built by *h*.anchor_bgn.

formatter

The *h*.formatter attribute is the formatter object *f* associated with *h*, which you pass as the only argument when you instantiate HTMLParser(*f*).

handle_image

```
h.handle_image(source,alt,ismap
='',align='',width='',height
='')
```

Called for each <img> tag. Each argument is the string value of the tag's attribute of the same name. HTMLParser's implementation of handle_image calls *h*.handle_data(*alt*).

nofill

```
h.nofill
```

The *h*.nofill attribute is false when the parser is collapsing whitespace, the normal case. It is true when the parser must

22.3 The HTMLParser Module

Module HTMLParser supplies one class, HTMLParser, that you subclass to override and add methods. HTMLParser.HTMLParser is similar to sgmllib.SGMLParser, but is simpler and able to parse XHTML as well. The main differences between HTMLParser and SGMLParser are the following:

- HMTLParser does not call back to methods named do_*tag*, start_*tag*, and end_*tag*. To process tags and end tags, your subclass *X* of HTMLParser must override methods handle_starttag and/or handle_endtag and check explicitly for the tags it wants to process.

- HMTLParser does not keep track of, nor check, tag nesting in any way.

- HMTLParser does nothing, by default, to resolve character and entity references. Your subclass *X* of HTMLParser must override methods handle_charref and/or handle_entityref if it needs to perform processing of such references.

The most frequently used methods of an instance *h* of a subclass *X* of HTMLParser are as follows.

close

`h.close(  )`

Tells the parser that there is no more input data. When *X* overrides close, *h*.close must also call HTMLParser.close to ensure that buffered data gets processed.

feed

`h.feed(data)`

Passes to the parser a part of the text being parsed. The parser processes some prefix of the text and holds the rest in a buffer until the next call to *h*.feed or *h*.close.

handle_charref

`h.handle_charref(ref)`

Called to process a character reference '&#*ref*;'. HTMLParser's implementation of handle_charref does nothing.

handle_comment

`h.handle_comment(comment)`

Called to handle comments. *comment* is the string within '<!--...-->', without the delimiters. HTMLParser's implementation of handle_comment does nothing.

22.4 Generating HTML

Python does not come with tools to generate HTML. If you want an advanced framework for structured HTML generation, I recommend Robin Friedrich's HTMLGen 2.2 (available at http://starship.python.net/crew/friedrich/HTMLgen/html/main.html), but I do not cover the package in this book. To generate XHTML, you can also use the approaches covered in Section 23.4 in Chapter 23.

22.4.1 Embedding

If your favorite approach is to embed Python code within HTML in the manner made popular by JSP, ASP, and PHP, one possibility is to use Python Server Pages (PSP) as supported by Webware, mentioned in Chapter 20. Another package, focused more specifically on the embedding approach, is Spyce (available at http://spyce.sf.net/). For all but the simplest problems, development and maintenance are eased by separating logic and presentation issues through templating, covered in the next section. Both Webware and Spyce optionally support templating in lieu of embedding.

22.4.2 Templating

To generate HTML, the best approach is often templating. With templating, you start with a *template*, which is a text string (often read from a file, database, etc.) that is valid HTML, but includes markers, also known as placeholders, where dynamically generated text must be inserted. Your program generates the needed text and substitutes it into the template. In the simplest case, you can use markers of the form '%(*name*)s'. Bind the dynamically generated text as the value for key '*name*' in some dictionary *d*. The Python string formatting operator %, covered in Chapter 9, now does all you need. If *t* is your template, *t%d* is a copy of the template with all values properly substituted.

22.4.3 The Cheetah Package

For advanced templating tasks, I recommend Cheetah (available at http://www.cheetahtemplate.org). Cheetah interoperates particularly well with Webware. When you have Webware installed, Cheetah's template objects are Webware servlets, so you can immediately deploy them under Webware. You can also use Cheetah in other contexts, and Spyce can also optionally use Cheetah for templating. Cheetah can process HTML templates for any purpose whatsoever. In fact, I recommend Cheetah to process templates for any kind of structured text, HTML or not.

22.4.3.1 The Cheetah templating language

In a Cheetah template, use $*name* or ${*name*} to request the insertion of the value of a variable named *name*. *name* can contain dots to request lookups of object attributes or dictionary keys. For example, $a.b.c requests insertion of the value of attribute c of attribute b of the variable named a. When b is a dictionary, this translates to the Python expression *a*.*b*['*c*']. If an object encountered during $ substitution is callable, Cheetah calls the object, without arguments, as a part of the lookup. This high degree of polymorphism makes authoring and maintaining Cheetah templates easier for non-developers, as it saves them the need to learn and understand these distinctions.

A Cheetah template can contain *directives*, which are verbs starting with # that allow comments, file inclusion, flow control (conditionals, loops, exception handling), and more. Cheetah basically provides a rich templating language on top of Python. The most frequently used verbs in simple Cheetah templates are the following (mostly similar to Python, but with $ in front of names, no trailing :, and no mandatory indents, but #end clauses instead):

#break, #continue, #pass

Chapter 23. Structured Text: XML

XML, the eXtensible Markup Language, has taken the programming world by storm over the last few years. Like SGML, XML is a metalanguage, a language to describe markup languages. On top of the XML 1.0 specification, the XML community (in good part inside the World Wide Web Consortium, W3C) has standardized other technologies, such as various schema languages, Namespaces, XPath, XLink, XPointer, and XSLT.

Industry consortia in many fields have defined industry-specific markup languages on top of XML, to facilitate data exchange among applications in the various fields. Such industry standards let applications exchange data even if the applications are coded in different languages and deployed on different platforms by different firms. XML, related technologies, and XML-based markup languages are the basis of interapplication, cross-language, cross-platform data interchange in modern applications.

Python has excellent support for XML. The standard Python library supplies the xml package, which lets you use fundamental XML technology quite simply. The third-party package PyXML (available at http://pyxml.sf.net) extends the standard library's xml with validating parsers, richer DOM implementations, and advanced technologies such as XPath and XSLT. Downloading and installing PyXML upgrades Python's own xml packages, so it can be a good idea to do so even if you don't use PyXML-specific features.

On top of PyXML, you can choose to install yet another freely available third-party package, 4Suite (available at http://4suite.org). 4Suite provides yet more XML parsers for special niches, advanced technologies such as XLink and XPointer, and code supporting standards built on top of XML, such as the Resource Description Framework (RDF).

As an alternative to Python's built-in XML support, PyXML, and 4Suite, you can try ReportLab's new pyRXP, a fast validating XML parser based on Tobin's RXP. pyRXP is DOM-like in that it constructs an in-memory representation of the whole XML document you're parsing. However, pyRXP does not construct a DOM-compliant tree, but rather a lightweight tree of Python tuples to save memory and enhance speed. For more information on pyRXP, see http://www.reportlab.com/xml/pyrxp.html.

For coverage of all aspects of XML and of how you can process XML with Python, I recommend Python & XML, by Christopher Jones and Fred Drake (O'Reilly). In this chapter, I cover only the essentials of the standard library's xml package, taking some elementary knowledge of XML itself for granted.

23.1 An Overview of XML Parsing

When your application must parse XML documents, your first, fundamental choice is what kind of parsing to use. You can use *event-driven* parsing, where the parser reads the document sequentially and calls back to your application each time it parses a significant aspect of the document (such as an element). Or you can use *object-based* parsing, where the parser reads the whole document and builds in-memory data structures, representing the document, that you can then navigate. SAX is the main, normal way to perform event-driven parsing, and DOM is the main, normal way to perform object-based parsing. In each case there are alternatives, such as direct use of expat for event-driven parsing and pyRXP for object-based parsing, but I do not cover these alternatives in this book. Another interesting possibility is offered by pulldom, which is covered later in this chapter.

Event-driven parsing requires fewer resources, which makes it particularly suitable when you need to parse very large documents. However, event-driven parsing requires you to structure your application accordingly, performing your processing (and typically building auxiliary data structures) in your methods that are called by the parser. Object-based parsing gives you more flexibility about the ways in which you can structure your application. It may be more suitable when you need to perform very complicated processing, as long as you can afford the extra resources needed for object-based parsing (typically, this means that you are not dealing with very large documents). Object-based approaches also support programs that need to modify or create XML documents, as covered later in this chapter.

As a general guideline, when you are still undecided after studying the various trade-offs, I suggest you try event-driven parsing when you can see a reasonably direct way to perform your program's tasks through this approach. Event-driven parsing is more scalable; therefore, if your program can perform its task via event-driven parsing, it will be applicable to larger documents than it would be able to handle otherwise. If event-driven parsing is too confining, try pulldom instead. I suggest you consider (non-pull) DOM only when you think DOM is the only way to perform your program's tasks without excessive contortions. In that case DOM may be best, as long as you can accept the resulting limitations, in terms of the maximum size of documents that your program is able to support and the costs in time and memory for processing.

23.2 Parsing XML with SAX

In most cases, the best way to extract information from an XML document is to parse the document with a parser compliant with SAX, the Simple API for XML. SAX defines a standard API that can be implemented on top of many different underlying parsers. The SAX approach to parsing has similarities to the HTML parsers covered in Chapter 22. As the parser encounters XML elements, text contents, and other significant events in the input stream, the parser calls back to methods of your classes. Such event-driven parsing, based on callbacks to your methods as relevant events occur, also has similarities to the event-driven approach that is almost universal in GUIs and in some networking frameworks. Event-driven approaches in various programming fields may not appear natural to beginners, but enable high performance and particularly high scalability, making them very suitable for high-workload cases.

To use SAX, you define a content handler class, subclassing a library class and overriding some methods. Then, you build a parser object *p*, install an instance of your class as *p*'s handler, and feed *p* the input stream to parse. *p* calls methods on your handler to reflect the document's structure and contents. Your handler's methods perform application-specific processing. The xml.sax package supplies a factory function to build *p*, as well as convenience functions for simpler operation in typical cases. xml.sax also supplies exception classes, used to diagnose invalid input and other errors.

Optionally, you can also register with parser *p* other kinds of handlers besides the content handler. You can supply a custom error handler to use an error diagnosis strategy different from normal exception raising, and try to diagnose several errors during a parse. You can supply a custom DTD handler to receive information about notation and unparsed entities from the XML document's Document Type Definition (DTD). You can supply a custom entity resolver to handle external entity references in advanced, customized ways. These additional possibilities are advanced and rarely used, so I do not cover them in this book.

23.2.1 The xml.sax Package

The xml.sax package supplies exception class SAXException, and subclasses of it to support fine-grained exception handling. xml.sax also supplies three functions.

make_parser

```
make_parser(parsers_list=[])
```

parsers_list is a list of strings, names of modules from which you would like to build your parser. make_parser tries each module in sequence until it finds one that defines a suitable function create_parser. After the modules in *parsers_list*, if any, make_parser continues by trying a list of default modules. make_parser terminates as soon as it can generate a parser *p*, and returns *p*.

parse

```
parse(file,handler,
error_handler=None)
```

file is a filename or a file-like object open for reading, containing an XML document. *handler* is generally an instance of your own subclass of class ContentHandler, covered later in this chapter. *error_handler*, if given, is generally an instance of your own subclass of class ErrorHandler. You don't necessarily have to subclass ContentHandler and/or ErrorHandler: you just need to provide the same interfaces as the classes do. Subclassing is often a convenient means to this end.

23.3 Parsing XML with DOM

SAX parsing does not build any structure in memory to represent the XML document. This makes SAX fast and highly scalable, as your application builds exactly as little or as much in-memory structure as needed for its specific tasks. However, for particularly complicated processing tasks involving reasonably small XML documents, you may prefer to let the library build in-memory structures that represent the whole XML document, and then traverse those structures. The XML standards describe the DOM (Document Object Model) for XML. A DOM object represents an XML document as a tree whose root is the *document object,* while other nodes correspond to elements, text contents, element attributes, and so on.

The Python standard library supplies a minimal implementation of the XML DOM standard, xml.dom.minidom. minidom builds everything up in memory, with the typical pros and cons of the DOM approach to parsing. The Python standard library also supplies a different DOM-like approach in module xml.dom.pulldom. pulldom occupies an interesting middle ground between SAX and DOM, presenting the stream of parsing events as a Python iterator object so that you do not code callbacks, but rather loop over the events and examine each event to see if it's of interest. When you do find an event of interest to your application, you can ask pulldom to build the DOM subtree rooted in that event's node by calling method expandNode, and then work with that subtree as you would in minidom. Paul Prescod, pulldom's author and XML and Python expert, describes the net result as "80% of the performance of SAX, 80% of the convenience of DOM." Other DOM parsers are part of the PyXML and 4Suite extension packages, mentioned at the start of this chapter.

23.3.1 The xml.dom Package

The xml.dom package supplies exception class DOMException and subclasses of it to support fine-grained exception handling. xml.dom also supplies a class Node, typically used as a base class for all nodes by DOM implementations. Class Node only supplies constant attributes giving the codes for node types, such as ELEMENT_NODE for elements, ATTRIBUTE_NODE for attributes, and so on. xml.dom also supplies constant module attributes with the URIs of important namespaces: XML_NAMESPACE, XMLNS_NAMESPACE, XHTML_NAMESPACE, and EMPTY_NAMESPACE.

23.3.2 The xml.dom.minidom Module

The xml.dom.minidom module supplies two functions.

parse

```
parse(file,parser=None)
```

file is a filename or a file-like object open for reading, containing an XML document. *parser*, if given, is an instance of a SAX parser class; otherwise, parse generates a default SAX parser by calling xml.sax.make_parser( ). parse returns a minidom document object instance representing the given XML document.

parseString

```
parseString(string,parser=None)
```

Like parse, except that *string* is the XML document in string form.

23.4 Changing and Generating XML

Just like for HTML and other kinds of structured text, the simplest way to output an XML document is often to prepare and write it using Python's normal string and file operations, covered in Chapter 9 and Chapter 10. Templating, covered in Chapter 22, is also often the best approach. Subclassing class XMLGenerator, covered earlier in this chapter, is a good way to generate an XML document that is like an input XML document, except for a few changes.

The xml.dom.minidom module offers yet another possibility, because its classes support methods to generate, insert, remove, and alter nodes in a DOM tree representing the document. You can create a DOM tree by parsing and then alter it, or you can create an empty DOM tree and populate it, and then output the resulting XML document with methods toxml, toprettyxml, or writexml of the Document instance. You can also output a subtree of the DOM tree by calling these methods on the Node that is the subtree's root.

23.4.1 Factory Methods of a Document Object

The Document class supplies factory methods to create new instances of subclasses of Node. The most frequently used factory methods of a Document instance *d* are as follows.

createComment

```
d.createComment(data)
```

Builds and returns an instance *c* of class Comment for a comment with text *data*.

createElement

```
d.createElement(tagname)
```

Builds and returns an instance *e* of class Element for an element with the given tag.

createTextNode

```
d.createTextNode(data)
```

Builds and returns an instance *t* of class TextNode for a text node with text *data*.

23.4.2 Mutating Methods of an Element Object

An instance *e* of class Element supplies the following methods to remove and add attributes.

removeAttribute

```
e.removeAttribute(name)
```

Removes *e*'s attribute with the given *name*.

Part V: Extending and Embedding

Chapter 24. Extending and Embedding Classic Python

Classic Python runs on a portable C-coded virtual machine. Python's built-in objects, such as numbers, sequences, dictionaries, and files, are coded in C, as are several modules in Python's standard library. Modern platforms support dynamic-load libraries, with file extensions such as *.dll* on Windows and *.so* on Linux, and building Python produces such binary files. You can code your own extension modules for Python in C, using the Python C API covered in this chapter, to produce and deploy dynamic libraries that Python scripts and interactive sessions can later use with the import statement, covered in Chapter 7.

Extending Python means building modules that Python code can import to access the features the modules supply. Embedding Python means executing Python code from your application. For such execution to be useful, Python code must in turn be able to access some of your application's functionality. In practice, therefore, embedding implies some extending, as well as a few embedding-specific operations.

Embedding and extending are covered extensively in Python's online documentation; you can find an in-depth tutorial at http://www.python.org/doc/ext/ext.html and a reference manual at http://www.python.org/doc/api/api.html. Many details are best studied in Python's extensively documented sources. Download Python's source distribution and study the sources of Python's core, C-coded extension modules and the example extensions supplied for study purposes.

This chapter covers the basics of extending and embedding Python with C. It also mentions, but does not cover, other possibilities for extending Python.

24.1 Extending Python with Python's C API

A Python extension module named *x* resides in a dynamic library with the same filename (*x.pyd* on Windows, *x.so* on most Unix-like platforms) in an appropriate directory (normally the *site-packages* subdirectory of the Python library directory). You generally build the *x* extension module from a C source file *x.c* with the overall structure:

```
#include <Python.h>

/* omitted: the body of the x module */

void
initx(void)
{
    /* omitted: the code that initializes the module named x */
}
```

When you have built and installed the extension module, a Python statement import *x* loads the dynamic library, then locates and calls the function named init*x*, which must do all that is needed to initialize the module object named *x*.

24.1.1 Building and Installing C-Coded Python Extensions

To build and install a C-coded Python extension module, it's simplest and most productive to use the distribution utilities, distutils, covered in Chapter 26. In the same directory as *x.c*, place a file named *setup.py* that contains at least the following statements:

```
from distutils.core import setup, Extension
setup(name='x', ext_modules=[ Extension('x',sources=['x.c']) ])
```

From a shell prompt in this directory, you can now run:

```
C:\> python setup.py install
```

to build the module and install it so that it becomes usable in your Python installation. The distutils perform all needed compilation and linking steps, with the right compiler and linker commands and flags, and copy the resulting dynamic library in an appropriate directory, dependent on your Python installation. Your Python code can then access the resulting module with the statement import *x*.

24.1.2 Overview of C-Coded Python Extension Modules

Your C function init*x* generally has the following overall structure:

```
void
initx(void)
{
    PyObject* thismod = Py_InitModule3("x", x_methods, "docstring for x");
    /* optional: calls to PyModule_AddObject(thismod, "somename", someobj)
       and other Python C API calls to finish preparing module object
       thismod and its types (if any) and other objects.
    */
}
```

More details are covered in Section 24.1.4 later in this chapter. *x_methods* is an array of PyMethodDef structs. Each PyMethodDef struct in the *x_methods* array describes a C function that your module *x* makes available to Python code that imports *x*. Each such C function has the following overall structure:

```
static PyObject*
func_with_named_arguments(PyObject* self, PyObject* args, PyObject* kwds)
{
    /* omitted: body of function, which accesses arguments via the Python C
```

24.2 Extending Python Without Python's C API

You can code Python extensions in other classic compiled languages besides C. For Fortran, the choice is between Paul Dubois's Pyfort (available at http://pyfortran.sf.net) and Pearu Peterson's F2PY (available at http://cens.ioc.ee/projects/f2py2e/). Both packages support and require the Numeric package covered in Chapter 15, since numeric processing is Fortran's typical application area.

For C++, the choice is between Gordon McMillan's simple, lightweight SCXX (available at http://www.mcmillan-inc.com/scxx.html), which uses no templates and is thus suitable for older C++ compilers, Paul Dubois's CXX (available at http://cxx.sf.net), and David Abrahams's Boost Python Library (available at http://www.boost.org/libs/python/doc). Boost is a package of C++ libraries of uniformly high quality for compilers that support templates well, and includes the Boost Python component. Paul Dubois, CXX's author, recommends considering Boost. You may also choose to use Python's C API from your C++ code, using C++ in this respect as if it was C, and foregoing the extra convenience that C++ affords. However, if you're already using C++ rather than C anyway, then using SCXX, CXX, or Boost can substantially improve your programming productivity when compared to using Python's C API.

If your Python extension is basically a wrapper over an existing C or C++ library (as many are), consider SWIG, the Simplified Wrapper and Interface Generator (available at http://www.swig.org). SWIG generates the C source code for your extension based on the library's header files, generally with some help in terms of further annotations in an interface description file.

Greg Ewing is developing a language, Pyrex, specifically for coding Python extensions. Pyrex (found at http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/) is an interesting mix of Python and C concepts, and is already quite usable despite being a new development.

The weave package (available at http://www.scipy.org/site_content/weave), lets you run inline C/C++ code within Python. The blitz function, in particular, generates and runs C++ code from expressions using the Numeric package, and thus requires Numeric.

If your application runs only on Windows, the most practical way to extend and embed Python is generally through COM. In particular, COM is by far the best way to use Visual Basic modules (packaged as ActiveX classes) from Python. COM is also the best way to make Python-coded functionality (packaged as COM servers) available to Visual Basic programs. The standard Python distribution for Windows does not directly support COM: you also need to download and install the platform-specific win32all extension package (available at http://starship.python.net/crew/mhammond/). I do not cover Windows-specific functionality, including COM, any further in this book. For excellent coverage of platform-specific Python use on Windows, I recommend Python Programming on Win32, by Mark Hammond and Andy Robinson (O'Reilly).

24.3 Embedding Python

If you have an application already written in C or C++ (or any other classic compiled language), you may want to embed Python as your application's scripting language. To embed Python in languages other than C, the other language must be able to call C functions. In the following, I cover only the C view of things, since other languages vary widely regarding what you have to do in order to call C functions from them.

24.3.1 Installing Resident Extension Modules

In order for Python scripts to communicate with your application, your application must supply extension modules with Python-accessible functions and classes that expose your application's functionality. If these modules are linked with your application rather than residing in dynamic libraries that Python can load when necessary, register your modules with Python as additional built-in modules by calling the PyImport_AppendInittab C API function.

PyImport_AppendInittab

```
int
PyImport_AppendInittab(char*
name,void (*initfunc)(void))
```

name is the module name, which Python scripts use in import statements to access the module. *initfunc* is the module initialization function, taking no argument and returning no result, as covered earlier in this chapter (i.e., *initfunc* is the module's function that would be named init*name* for a normal extension module residing in a dynamic library). PyImport_AppendInittab must be called before calling Py_Initialize.

24.3.2 Setting Arguments

You may want to set the program name and arguments, which Python scripts can access as sys.argv, by calling either or both of the following C API functions.

Py_SetProgramName

```
void Py_SetProgramName(char*
name)
```

Sets the program name, which Python scripts can access as sys.argv[0]. Must be called before calling Py_Initialize.

PySys_SetArgv

```
void PySys_SetArgv(int argc
,char** argv)
```

Sets the program arguments, which Python scripts can access as sys.argv[1:]. Must be called after calling Py_Initialize.

24.3.3 Python Initialization and Finalization

After installing extra built-in modules and optionally setting the program name, your application initializes Python. At

Chapter 25. Extending and Embedding Jython

Jython implements Python on a Java Virtual Machine (JVM). Jython's built-in objects, such as numbers, sequences, dictionaries, and files, are coded in Java. To extend Classic Python with C, you code C modules using the Python C API (as covered in Chapter 24). To extend Jython with Java, you do not have to code Java modules in special ways: every Java package on the Java CLASSPATH (or on Jython's sys.path) is automatically available to your Jython scripts and Jython interactive sessions for use with the import statement covered in Chapter 7. This applies to Java's standard libraries, third-party Java libraries you have installed, and Java classes you have coded yourself. You can also extend Java with C using the Java Native Interface (JNI), and such extensions will also be available to Jython code, just as if they had been coded in pure Java rather than in JNI-compliant C.

For details on advanced issues related to interoperation between Java and Jython, I recommend Jython Essentials, by Samuele Pedroni and Noel Rappin (O'Reilly). In this chapter, I offer a brief overview of the simplest interoperation scenarios, which suffices for a large number of practical needs. Importing, using, extending, and implementing Java classes and interfaces in Jython just works in most practical cases of interest. In some cases, however, you need to be aware of issues related to accessibility, type conversions, and overloading, as covered in this chapter. Embedding the Jython interpreter in Java-coded applications is similar to embedding the Python interpreter in C-coded applications (as covered in Chapter 24), but the Jython task is easier. Jython offers yet another possibility for interoperation with Java, using the *jythonc* compiler to turn your Python sources into classic, static JVM bytecode *.class* and *.jar* files. You can then use these bytecode files in Java applications and frameworks, exactly as if their source code had been in Java rather than in Python.

25.1 Importing Java Packages in Jython

Unlike Java, Jython does not implicitly and automatically import java.lang. Your Jython code can explicitly import java.lang, or even just import java, and then use classes such as java.lang.System and java.lang.String as if they were Python classes. Specifically, your Jython code can use imported Java classes as if they were Python classes with a _ _slots_ _ class attribute (i.e., you cannot create arbitrary new instance attributes). You can subclass a Java class with your own Python class, and instances of your class let you create new attributes just by binding them, as usual.

You may choose to import a top-level Java package (such as java) rather than specific subpackages (such as java.lang). Your Python code acquires the ability to access all subpackages when you import the top-level package. For example, after import java, your code can use classes java.lang.String, java.util.Vector, and so on.

The Jython runtime wraps every Java class you import in a transparent proxy, which manages communication between Python and Java code behind the scenes. This gives an extra reason to avoid the dubious idiom from *somewhere* import *, in addition to the reasons mentioned in Chapter 7. When you perform such a bulk import, the Jython runtime must build proxy wrappers for all the Java classes in package *somewhere*, spending substantial amounts of memory and time wrapping classes your code will probably not use. Avoid from ... import * except for occasional convenience in interactive exploratory sessions, and stick with the import statement. Alternatively, it's okay to use specific, explicit from statements for classes you know your Python code wants to use (e.g., from java.lang import System).

25.1.1 The Jython Registry

Jython relies on a *registry* of Java properties as a cross-platform equivalent of the kind of settings that would normally use the Windows registry, or environment variables on Unix-like systems. Jython's registry file is a standard Java properties file named registry, located in a directory known as the Jython root directory. The Jython root directory is normally the directory where jython.jar is located, but you can override this by setting Java properties python.home or install.root. For special needs, you may tweak the Jython registry settings via an auxiliary Java properties file named .jython in your home directory, and/or via command-line options to the *jython* interpreter command. The registry option python.path is equivalent to classic Python's PYTHONPATH environment variable. This is the option you may most often be interested in, as it can help you install extra Python packages outside of the Jython installation directories (e.g., sharing Python packages installed for CPython use).

25.1.2 Accessibility

Normally, your Jython code can access only public features (methods, fields, inner classes) of Java classes. You may choose to make private and protected features available by setting an option in the Jython registry before you run Jython:

```
python.security.respectJavaAccessibility=false
```

Such bending of normal Java rules should never be necessary for normal operation. However, the ability to access private and protected features may be useful to Jython scripts meant to thoroughly test a Java package, which is why Jython gives you this option.

25.1.3 Type Conversions

The Jython runtime converts data between Python and Java transparently. However, when a Java method expects a boolean argument, you have to pass an int or an instance of java.lang.Boolean in order to call that method from

25.2 Embedding Jython in Java

Your Java-coded application can embed the Jython interpreter in order to use Jython for scripting. jython.jar must be in your Java CLASSPATH. Your Java code must import org.python.core.* and org.python.util.* in order to access Jython's classes. To initialize Jython's state and instantiate an interpreter, use the Java statements:

```
 PySystemState.initialize(  );
PythonInterpreter interp = new PythonInterpreter(  );
```

Jython also supplies several advanced overloads of this method and constructor in order to let you determine in detail how PySystemState is set up, and to control the system state and global scope for each interpreter instance. However, in typical, simple cases, the previous Java code is all your application needs.

25.2.1 The PythonInterpreter Class

Once you have an instance *interp* of class PythonInterpreter, you can call method *interp*.eval to have the interpreter evaluate a Python expression held in a Java string. You can also call any of several overloads of *interp*.exec and *interp*.execfile to have the interpreter execute Python statements held in a Java string, a precompiled Jython code object, a file, or a Java InputStream.

The Python code you execute can import your Java classes in order to access your application's functionality. Your Java code can set attributes in the interpreter namespace by calling overloads of *interp*.set, and get attributes from the interpreter namespace by calling overloads of *interp*.get. The methods' overloads give you a choice. You can work with native Java data and let Jython perform type conversions, or you can work directly with PyObject, the base class of all Python objects, covered later in this chapter. The most frequently used methods and overloads of a PythonInterpreter instance *interp* are the following.

eval

```
PyObject interp.eval(String s)
```

Evaluates, in *interp*'s namespace, the Python expression held in Java string *s*, and returns the PyObject that is the expression's result.

exec

```
void interp.exec(String s)
void interp.exec(PyObject code)
```

Executes, in *interp*'s namespace, the Python statements held in Java string *s* or in compiled PyObject *code* (produced by function _ _builtin_ _.compile of package org.python.core, covered later in this chapter).

execfile

```
void interp.execfile(String
name)
void interp
.execfile(java.io.InputStream s
)
void interp
.execfile(java.io.InputStream s
,String name)
```

25.3 Compiling Python into Java

Jython comes with the *jythonc* compiler. You can feed *jythonc* your *.py* source files, and *jythonc* compiles them into normal JVM bytecode and packages them into *.class* and *.jar* files. Since *jythonc* generates static, classic bytecode, it cannot quite cope with the whole range of dynamic possibilities that Python allows. For example, *jythonc* cannot successfully compile Python classes that determine their base classes dynamically at runtime, as the normal Python interpreters allow. However, except for such extreme examples of dynamically changeable class structures, *jythonc* does support compilation of essentially the whole Python language into Java bytecode.

25.3.1 The jythonc command

jythonc resides in the Tools/jythonc directory of your Jython installation. You invoke it from a shell (console) command line with the syntax:

```
jythonc options modules
```

options are zero or more option flags starting with --. *modules* are zero or more names of Python source files to compile, either as Python-style names of modules residing on Python's sys.path, or as relative or absolute paths to Python source files. Include the *.py* extension in each path to a source file, but not in a module name.

More often than not, you will specify the *jythonc* option --jar *jarfile*, to build a *.jar* file of compiled bytecode rather than separate *.class* files. Most other options deal with what to put in the *.jar* file. You can choose to make the file self-sufficient (for browsers and other Java runtime environments that do not support using multiple *.jar* files) at the expense of making the file larger. Option --all ensures all Jython core classes are copied into the *.jar* file, while --core tries to be more conservative, copying as few core classes as feasible. Option --addpackages *packages* lets you list (in *packages*, a comma-separated list) those external Java packages whose classes are copied into the *.jar* file if any of the Python classes *jythonc* is compiling depends on them. An important alternative to --jar is --bean *jarfile*, which also includes a bean manifest in the *.jar* file as needed for Python-coded JavaBeans components.

Another useful *jythonc* option is --package *package*, which instructs Jython to place all the new Java classes it's creating in the given *package* (and any subpackages of *package* needed to reflect the Python-side package structure).

25.3.2 Adding Java-Visible Methods

The Java classes that *jythonc* creates normally extend existing classes from Java libraries and/or implement existing interfaces. Other Java-coded applications and frameworks instantiate the *jythonc*-created classes via constructor overloads, which have the same signatures as the constructors of their Java superclasses. The Python-side __init__ executes after the superclass is initialized, and with the same arguments (therefore, don't __init__ a Java superclass in the __init__ of a Python class meant to be compiled by *jythonc*). Afterward, Java code can access the functionality of instances of Python-coded classes by calling instance methods defined in known interfaces or superclasses and overridden by Python code.

Python code can never supply Java-visible static methods or attributes, only instance methods. By default, each Python class supplies only the instance methods it inherits from the Java class it extends or the Java interfaces it implements. However, Python code can also supply other Java-visible instance methods via the @sig directive.

To expose a method of your Python class to Java when *jythonc* compiles the class, code the method's docstring as @sig followed by a Java method signature. For example:

Chapter 26. Distributing Extensions and Programs

Python's distutils allow you to package Python programs and extensions in several ways, and to install programs and extensions to work with your Python installation. As I mentioned in Chapter 24, the distutils also afford the most effective way to build C-coded extensions you write yourself, even when you are not interested in distributing such extensions. This chapter covers the distutils, as well as third-party tools that complement the distutils and let you package Python programs for distribution as standalone applications, installable on machines with specific hardware and operating systems without a separate installation of Python.

26.1 Python's distutils

The distutils are a rich and flexible set of tools to package Python programs and extensions for distribution to third parties. I cover typical, simple use of the distutils for the most common packaging needs. For in-depth, highly detailed discussion of distutils, I recommend two manuals that are part of Python's online documentation: Distributing Python Modules (available at http://www.python.org/doc/current/dist/), and Installing Python Modules (available at http://www.python.org/doc/current/inst/), both by Greg Ward, the principal author of the distutils.

26.1.1 The Distribution and Its Root

A *distribution* is the set of files to package into a single file for distribution purposes. A di stribution may include zero, one, or more Python packages and other Python modules (as covered in Chapter 7), as well as, optionally, Python scripts, C-coded (and other) extensions, supporting data files, and auxiliary files containing metadata about the distribution itself. A distribution is said to be *pure* if all code it includes is Python, and *non-pure* if it also includes non-Python code (most often, C-coded extensions).

You should normally place all the files of a distribution in a directory, known as the distribution root directory, and in subdirectories of the distribution root. Mostly, you can arrange the subtree of files and directories rooted at the distribution root to suit your own organizational needs. However, remember from Chapter 7 that a Python package must reside in its own directory, and a package's directory must contain a file named *_ _init_ _.py* (or subdirectories with *_ _init_ _.py* files, for subpackages) as well as other modules belonging to that package.

26.1.2 The setup.py Script

The distribution root directory must contain a Python script that by convention is named *setup.py*. The *setup.py* script can, in theory, contain arbitrary Python code. However, in practice, *setup.py* always boils down to some variation of:

```
 from distutils.core import setup, Extension

setup( many keyword arguments go here )
```

All the action is in the parameters you supply in the call to setup. You should not import Extension if your *setup.py* deals with a pure distribution. Extension is needed only for non-pure distributions, and you should import it only when you need it. It is fine to have a few statements before the call to setup, in order to arrange setup's arguments in clearer and more readable ways than could be managed by having everything inline as part of the setup call.

The distutils.core.setup function accepts only keyword arguments, and there are a large number of such arguments that you could potentially supply. A few deal with the internal operations of the distutils themselves, and you never supply such arguments unless you are extending or debugging the distutils, an advanced subject that I do not cover in this book. Other keyword arguments to setup fall into two groups: metadata about the distribution, and information about what files are in the distribution.

26.1.3 Metadata About the Distribution

You should provide metadata about the distribution by supplying some of the following keyword arguments when you call the distutils.core.setup function. The value you associate with each argument name you supply is a string that is intended mostly to be human-readable; therefore, any specifications about the string's format are just advisory. The explanations and recommendations about the metadata fields in the following are also non-normative, and correspond only to common, not universal, conventions. Whenever the following explanations refer to "this distribution," it can be taken to refer to the material included in the distribution, rather than to the packaging of the distribution.

26.2 The py2exe Tool

The distutils help you package up your Python extensions and applications. However, an end user can install the resulting packaged form only after installing Python. This is particularly a problem on Windows, where end users want to run a single installer to get an application working on their machine. Installing Python first and then running your application's installer may prove too much of a hassle for such end users.

Thomas Heller has developed a simple solution, a distutils add-on named py2exe, freely available for download from http://starship.python.net/crew/theller/py2exe/. This URL also contains detailed documentation of py2exe, and I recommend that you study that documentation if you intend to use py2exe in advanced ways. However, the simplest kinds of use, which I cover in the rest of this section, cover most practical needs.

After downloading and installing py2exe (on a Windows machine where Microsoft Visual C++ 6 is also installed), you just need to add the line:

```
import py2exe
```

at the start of your otherwise normal distutils script *setup.py*. Now, in addition to other distutils commands, you have one more option. Running:

```
python setup.py py2exe
```

builds and collects in a subdirectory of your distribution root directory an *.exe* file and one or more *.dll* files. If your distribution's name metadata is, for example, myapp, then the directory into which the *.exe* and *.dll* files are collected is named *dist\myapp\*. Any files specified by option data_files in your *setup.py* script are placed in subdirectories of *dist\myapp\*. The *.exe* file corresponds to your application's first or single entry in the scripts keyword argument value, and also contains the bytecode-compiled form of all Python modules and packages that your *setup.py* specifies or implies. Among the *.dll* files is, at minimum, the Python dynamic load library, for example *python22.dll* if you use Python 2.2, plus any other *.pyd* or *.dll* files that your application needs, excluding *.dll* files that py2exe knows are system files (i.e., guaranteed to be available on any Windows installation).

py2exe provides no direct means to collect the contents of the *dist\myapp\* directory for easy distribution and installation. You have several options, ranging from a *.zip* file (which may be given an *.exe* extension and made self-extracting, in ways that vary depending on the *.zip* file handling tools you choose), all the way to a professional Windows installer construction system, such as those sold by companies such as Wise and InstallShield. One option that is particularly worth considering is Inno Setup, a free, professional-quality installer construction system (see http://www.jrsoftware.org/isinfo.php). Since the files to be packaged up for end user installation are an *.exe* file, one or more *.dll* files, and perhaps some data files in subdirectories, the issue becomes totally independent from Python. You may package up and redistribute such files just as if they had originally been built from sources written in any other programming language.

26.3 The Installer Tool

Gordon McMillan has developed a richer and more general solution to the same problem that py2exe solves—preparing compact ways to package up Python applications for installation on end user machines that may not have Python installed. The Installer tool, freely downloadable from http://www.mcmillan-inc.com/installer, is more general than py2exe, which supports only Windows platforms. Installer natively supports Linux as well as Windows. Also, Installer's portable, cross-platform architecture may allow you to extend it to support other Unix-like platforms with a reasonable amount of effort.

Installer does not rely on distutils. To use Installer, you must learn its own specification files' syntax and semantics. Installer can do much more than py2exe, so it's not surprising that there is more for you to learn before making full use of it. However, I recommend studying and trying out Installer if you have the specific need of building standalone Python applications for Linux or other Unix-like architectures, or if you have tried py2exe and found it did not quite meet your needs.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of Python in a Nutshell is an African rock python, one of approximately 18 species of python. Pythons are nonvenomous constrictor snakes that live in tropical regions of Africa, Asia, Australia, and some Pacific Islands. Pythons live mainly on the ground, but they are also excellent swimmers and climbers. Both male and female pythons retain vestiges of their ancestral hind legs. The male python uses these vestiges, or spurs, when courting a female.

The python kills its prey by suffocation. While the snake's sharp teeth grip and hold the prey in place, the python's long body coils around its victim's chest, constricting tighter each time it breathes out. They feed primarily on mammals and birds. Python attacks on humans are extremely rare.

Emily Quill was the production editor and copyeditor for Python in a Nutshell. Linley Dolby and Tatiana Apandi Diaz provided quality control. Philip Dangler, Judy Hoer, and Genevieve d'Entremont provided production assistance. Nancy Crumpton wrote the index.

Emma Colby designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

Bret Kerr designed the interior layout, based on a series design by David Futato. This book was converted by Mike Sierra to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. This colophon was written by Nicole Arigo.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Madeleine Newell) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, and Jeff Liggett.

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z
]

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z
]

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z
]